A Systolic Algorithm for VLSI Design Of a Veterbi Decoder

by

Ali Fiqhi Damati

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

ELECTRICAL ENGINEERING

June, 1988

Order Number 1355731

A systolic algorithm for VLSI design of a Viterbi decoder

Damati, Ali Fiqhi Mohammad, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A SYSTOLIC ALGORITHM FOR VLSI DESIGN OF A VITERBI DECODER

BY

ALI FIQHI DAMATI

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
ELECTRICAL ENGINEERING

JUNE 1988

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by ALI FIQHI M. A. DAMATI under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in ELECTRICAL ENGINEERING

THESIS COMMITTEE

Thesis Advisor

Member

Member

Member

Department Chairman

Dean, College of Graduate Studies

August 8, 1988
Date

This thesis is dedicated to my parents

ACKNOWLEDGMENT

Acknowledgment is due to the King Fahd University of Petroleum and Minerals. Acknowledgment is also due to Department of Electrical Engineering for providing the opportunity to carry out this research work, and to the Department of Computer Engineering for granting me access to their computing facilities.

I gratefully acknowledge the tremendous help, guidance and encouragement by Dr. Mushfiqur Rahman who served as my committee chairman. I wish to express my thanks and appreciation to Dr. Sadiq M. Sait, the committee co-chairman, for introducing me to systolic arrays and for his valuable assistance throughout the research. I am grateful to the other two committee members, Dr. Gerhard F. Beckhoff and Dr. Essam Hassan for their useful comments and corrections.

I express my appreciations for the care and help received from my friends, particularly Anan Yaagoub, Ahmad Ghazal, Yahya Garout, Khalid Hamied, Tharwat Al-Jamaan, Mohammad Al-Samman and Yousef Dhahir.

Lastly, and in no sense the least, I am thankful to all faculty, colleagues and friends who made my stay at the university a memorable and valuable experience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THESIS ABSTRACT

NAME OF STUDENT: ALI FIQHI MOHAMMAD DAMATI

TITLE OF STUDY : A SYSTOLIC ALGORITHM FOR VLSI DESIGN OF A VITERBI DECODER

MAJOR FIELD : ELECTRICAL ENGINEERING

DATE OF DEGREE : JUNE 1988

The large number of potential applications of the Viterbi algorithm has given a great motivation for a thorough investigation of possible means of implementing it using present state technology. The draw back of Viterbi decoding, however, is the complexity of the decoder hardware. In this research an algorithm has been exploited further to simplify hardware implementation.

A powerful methodology for mapping high-level computations into hardware structure is the systolic arrays. In this research a new systolic algorithm for decoding convolutional codes using the Viterbi maximum likelihood decoding is presented. The new algorithm has been developed in order to design a high-performance, low-cost Viterbi decoder. CMOS design for basic cells for a 1/2 rate Viterbi decoder have been developed. Analysis of the proposed systolic Viterbi decoder has shown that the new decoder is superior to the existing ones in terms of speed, area, and power dissipation. The new design is versatile and has the capability for extending to any R=k/n rate decoder.

MASTER OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

Dhahran, Saudi Arabia

June 1988

خلاصة الرسالة

اسم الطالـــب : علي فقهي محمد دماطي
عنوان الدراسة : خوارزميه توافقيه لتصميم محلل فيتيربي للرموز الشفريه ذا مستوى عالٍ جداً من الادماج
التخصـــــص : هندسة كهربائيه
تاريخ الشهادة : حزيران ١٩٨٨ م

إن العدد الهائل من التطبيقات المعتمدة على خوارزميه فيتيربي أعطى دافعا كبيراً للبحث عن الطرق الممكنه لتنفيذ هذه الخوارزميه باستخدام التقينه الحديثه . لكن العائق الوحيد لحل الشفرات الالتفافيه باستخدام خوارزميه فيتيربي هو مدى تعقيد تصميم محلل الشفرات . في هذا البحث نقدم خوارزميه جديده لتسهيل تنفيذ دوائر المحللات .

من الطرق الفعاله لتحويل الحسابات ذات المستوى العالي من التعقيد لبنيه قابله للتنفيذ هي الطريقة المسماة بالصفوف التوافقيه (الانقباضيه) في هذا البحث تعرض خوارزميه توافقيه جديدة لتسهيل حل الشفرات الالتفافيه باستخدام محلل فيتيربي للرموز الشفريه .

وقد طورت هذه الخوارزميه الجديدة لتجعل بالامكاينه تصميم محللات فيتيربي للرموز الشفريه ذات أداء عال للغايه وبأقل التكاليف . كذلك ثم تصميم الخلايا الاساسيه لمحلل الشفرات ذات النسبه النصفيه باستخدام تقنيه CMOS . ولقد دلت الابحاث التى اجريت على محلل الشفرات التوافقي الجديد أنه يتفوق بدرجة كبيره للغايه على المحللات الموجودة حاليًا من حيث السرعه والحجم والطاقه المنبعثه . كذلك فإن التصميم الجديد يتميز بأنه قابل للتطوير ليناسب أي نسبه كسريه من الشفرات الالتفافيه .

درجة الماجستير في العلوم
جامعة الملك فهد للبترول المعادن
الظهران - المملكة العربية السعودية
حزيران ١٩٨٨ م

CHAPTER I

INTRODUCTION

The growth of microelectronics and semiconductor technology has given a great motivation towards the field of very large scale integration (VLSI). Many strategies have been followed in order to improve VLSI designs. Systolic arrays is a recent methodology for mapping high-level computations into hardware structure.

In this thesis a systolic algorithm has been developed in order to design high-performance, low-cost architecture for decoding convolutional codes using Viterbi algorithm. Strategy flow of the design followed is depicted in Figure 1.1.

1.1 STATEMENT OF THE PROBLEM

The large number of potential applications of the Viterbi algorithm provided sufficient motivation for a thorough investigation of possible means of implementing it using present state technology. The draw back of Viterbi decoding, however, is the complexity of the decoder

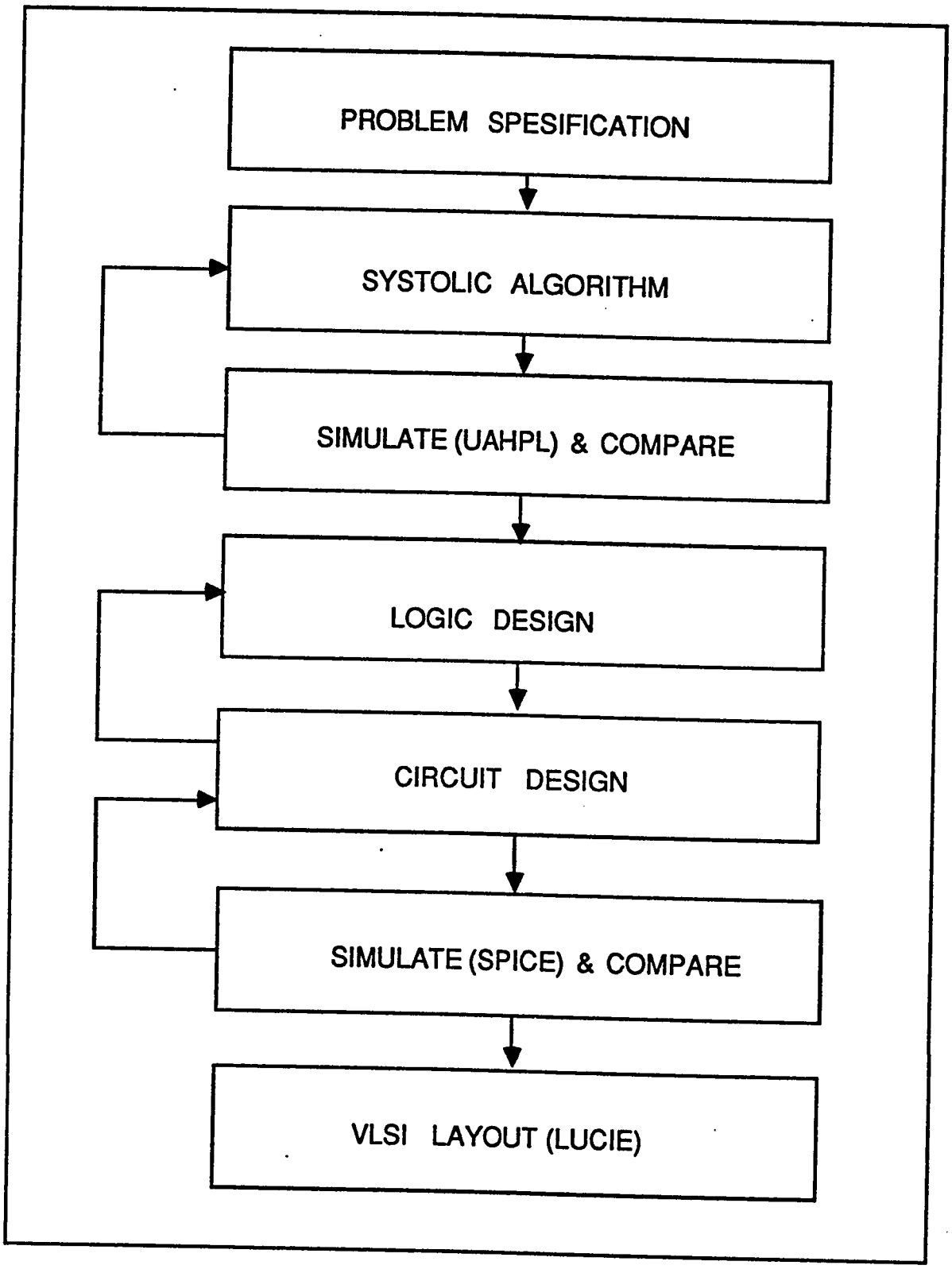


Figure 1.1 : Decoder design strategy

hardware which is due to the large storage required. Also the number of computations in a decoding process increases exponentially with the constraint length of the code. The complexity of the Viterbi decoder has always been a problem, preventing the algorithm from being fully exploited.

Many architectures for Viterbi decoders have been proposed in the last two decades. Most architectures proposed are either microprocessor-based or hard-wired designs using SSI and MSI technology. The problem of the above mentioned architectures is that they occupy large areas, and that large amount of power is dissipated. These disadvantages limit their applicabilities in some communication systems such as space communication systems. Communication from a distant and isolated object in space to a ground-base station presents certain system problems which are not nearly as critical in earth-based communication systems. The most obvious among these is the high cost of space-platform power [17]. It is desirable to design decoders which are as efficient as practical, and their power consumption and weight are as small as possible.

In this thesis a systolic algorithm, for decoding convolutional codes using the Viterbi maximum likelihood decoding, is proposed. Systolic algorithms schedule computations in such a way that a data item is not only used when it is input but is also reused as it moves through the pipelines in the array. The systolic algorithm is proposed so as to be

used for VLSI implementation. The tendency of systolic algorithms to provide simpler, regular, and modular layouts make them well-suited for VLSI implementation. Systolic arrays attempt to meet topological constraints mentioned above by using simple processors with a simple interconnection pattern. CMOS semiconductor technology has been used to provide a new architecture for Viterbi decoders. Reason behind using CMOS technology is to provide a system where power dissipation is minimized and chip area occupied is reduced.

1.2 THESIS OBJECTIVES

Thesis objectives include the following:

1) Development of new systolic algorithm for Viterbi decoding. The resulting algorithm should reflect the properties of systolic arrays as powerful computing systems.

2) Simulation of the systolic system using UAHPL (Universal Hardware Programming Language) [18]. The UAHPL simulation can be used to make sure that the systolic design is working properly and to get rid of any bugs in the design.

3) Design of basic cells of a 1/2 rate systolic Viterbi decoder using CMOS technology. These cells are to be used in a VLSI design of the decoder.

4) Extension of the decoder design to any R=k/n rate and

suggestions of possible modifications for the general case (R=k/n rate).

5) Investigation of comparison of this systolic design with others regarding storage requirement, and complexity of hardware.

6) Investigation of the different design techniques that could enhance the performance of the systolic system, and suggestions for future work.

1.3 THESIS ORGANIZATION

The concepts of convolutional codes, their structures, and Viterbi maximum likelihood decoding algorithm are introduced in chapter 2. Chapter 3 discusses systolic arrays and their design methodology. In chapter 4 the new proposed systolic algorithm for a Viterbi decoder is discussed in detail. An architecture of a 1/2 rate Viterbi decoder is also given. An RTL model and UAHPL simulation outputs for a Viterbi decoder are provided at the end of chapter 4. Chapter 5 deals with the design of basic cells of the decoder. In chapter 6 an analysis and performance evaluation of the systolic Viterbi decoder are presented. Conclusions and suggested future work are given in chapter 7.

CHAPTER II

CONVOLUTIONAL CODES AND VITERBI DECODING

2.1 INTRODUCTION

Error-correction coding is used in digital data communication in order to improve the reliability of communication over digital channels. There are two families of codes in common use today, (n,k) block codes and (n,k,m) convolutional codes. Both codes map a k-symbol input sequence into an n-symbol output sequence. In block codes the output is determined by the current k-data digits that are accumulated and then encoded into an n-digit code word. In convolutional codes the output is determined by the current input and a span of (m-1) preceding inputs where m is defined as the "constraint length" of the code.

Convolutional codes have been proved [10] to be equal to or superior than block codes in performance and are generally simpler than comparable block codes in implementation. Convolutional coding techniques were first introduced by Elias [9] in 1955. Since then many

algorithms to decode convolutional codes were introduced. The three most widely used decoding techniques are Viterbi maximum likelihood decoding, syndrome decoding, and sequential decoding. Each technique posesses special advantages over others. For example, feedback decoders, which fall in the syndrome decoding techniques, are well suited to correcting error bursts which may occur in fading channels [36], while sequential decoding techniques are very attractive for convolutional codes whose constraint length is quite large (m>9) [36]. Long constraint lengths are necessary for extremely low error probability. Sequential decoding algorithm in fact, necessitates m to be large in order to be able to find the correct path. The main drawback of sequential decoding algorithm is that the number of incorrect path branches, and consequently the computation complexity, is a random variable depending on the channel noise [36].

In this chapter the structure of the convolutional encoder is briefly viewed. Viterbi algorithm (VA) for decoding convolutional codes is discussed in detail. An illustrative example for decoding using VA is provided.

2.2 STRUCTURE OF CONVOLUTIONAL CODES

A constraint length m convolutional encoder is a linear finite state machine consisting of an m-stage shift register and n linear algebraic function generators. Each generator is a function of some of the shift

register stages. The generators are connected to a commutator which scans their outputs.

Information symbols are shifted in along the shift register k bits at a time (see Figure 2.1). A simple example of a convolutional encoder is the 1/2 rate encoder shown in Figure 2.2 where the constraint length m = 3, n=2 and k=1. The connections between the shift register stages and the modulo-2 address can be described by generator polynomials. The polynomials $g_1(x) = 1 + x + x^2$ and $g_2(x) = 1 + x^2$ represent the upper and lower connections, respectively. The input information sequence can also be represented as a power series

$$I(x) = i_0 + i_1 x + i_2 x^2 + \text{-----}$$

where, i_j is the information symbol (0 or 1) at the jth input symbol time. The outputs (Y_1 and Y_2) of the convolutional encoder can be described as a polynomial multiplication of the input sequence I and the generator polynomials g_1 and g_2. In our example

$Y_1(x) = I(x)g_1(x)$ and $Y_2(x) = I(x)g_2(x)$

where the polynomial multiplication is carried out in GF(2).

Another convenient way of describing the relationship between input and output sequences is the tree structure. Figure 2.3 shows a tree-code representation for the encoder of Figure 2.2 There are 2^k out branches for each node. Each branch carries n coded symbols

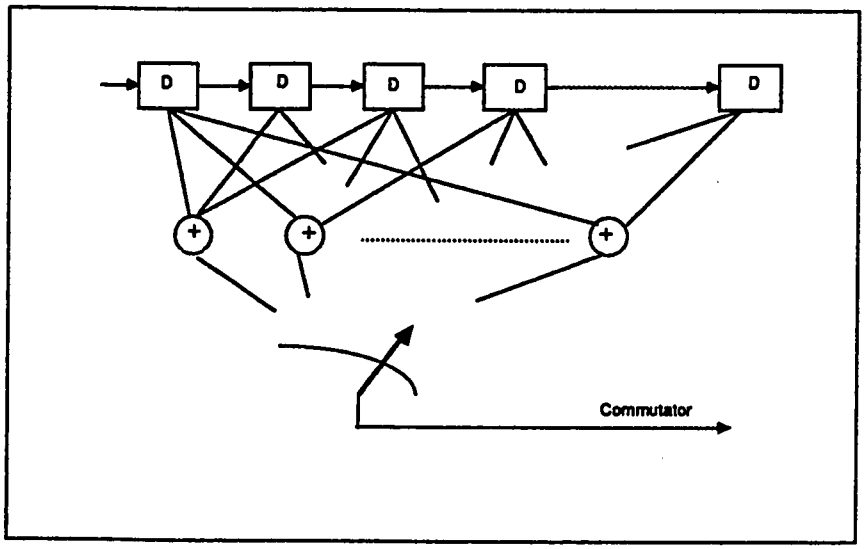


Figure 2.1: Convolutional encoder

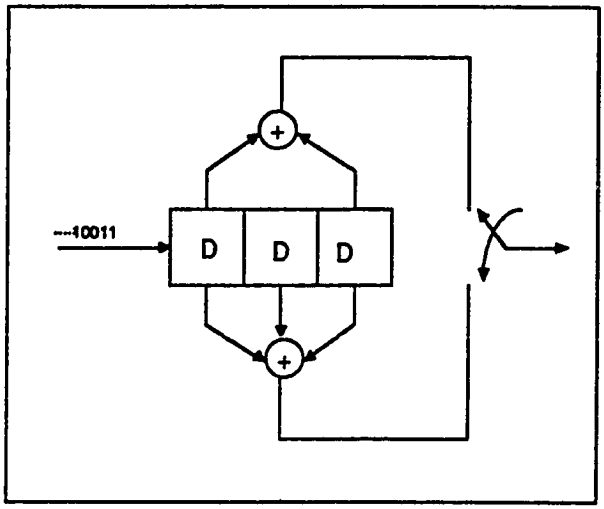


Figure 2.2: 1/2 rate convolutional encoder with m=3

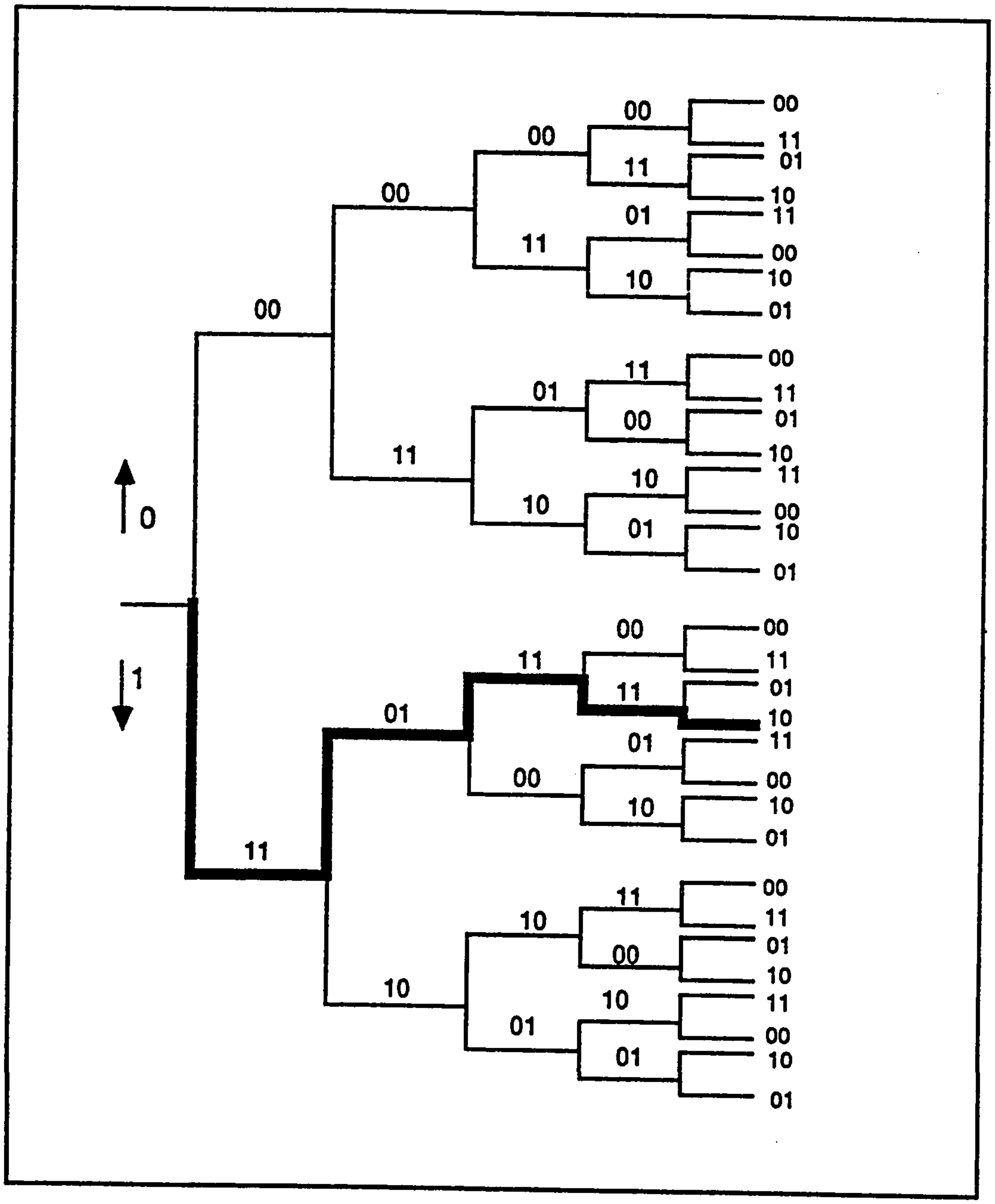


Figure 2.3: Tree-code representation

corresponding to a single input symbol. A path in the tree is traced according to the input sequence that specifies it. For the binary tree of Figure 2.3 the convention used is that an input '0' corresponds to the upper branch and an input '1' corresponds to the lower branch. For example, an input sequence of 1 0 0 1 1 causes a 11 01 11 11 01 output sequence and the path traced is indicated in Figure 2.3 by a thick line. From the structure of the tree-code representation it is obvious that a sequence of n information digits entering the encoder will correspond to one of the possible r^n (r represents the alphabet size) paths in the tree which indicates an exponential growth of the code tree with respect to the input sequence.

Fortunately, the structure of a tree associated with a convolutional code becomes repetitive after the number of branches equals the constraint length m. By taking this into account and merging equivalent branches, a new simplified tree diagram, called a trellis structure can be drawn. A trellis structure is composed of r^{m-k} nodes at each time step where r represents the alphabet size (r=2 for binary trellis). Each node of the trellis structure represents a distinct state at a given time, while every branch, going from a certain node, of the structure represents a transition to some new state at the next time instant. Figure 2.4 depicts the trellis structure corresponding to the tree-code representation shown in Figure 2.3

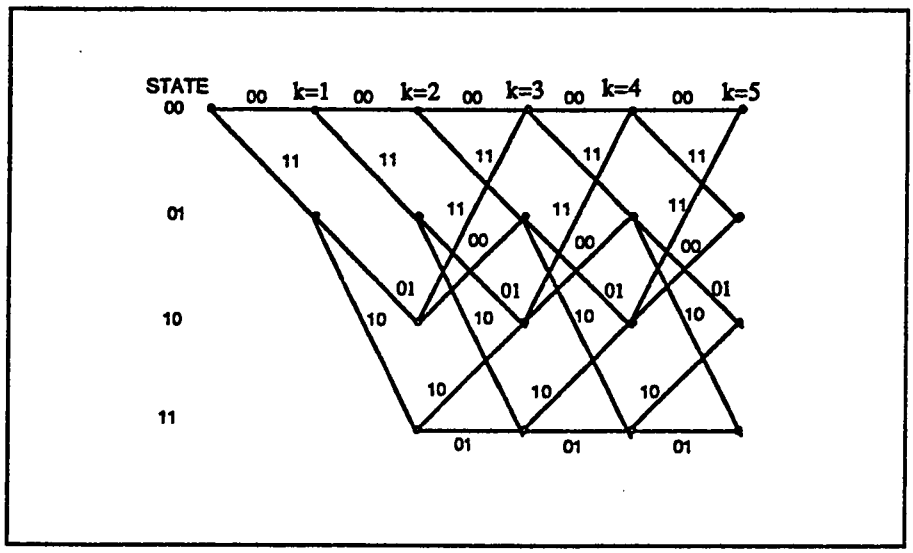


Figure 2.4: Trellis Structure for constraint length 3

The completely repetitive structure of the trellis diagram suggests a further reduction in the representation of the code to the so called the state diagram. The state diagram can be regarded as a signal flow graph. Each node of the state diagram represents a state, while an arrow from a certain state to any other state represents a transition to that state provided that the input shown on the arrow is processed. Figure 2.5 shows the state diagram corresponding to the trellis structure of Figure 2.4.

As we will show later, the trellis structure plays a significant role in decoding convolutional codes using Viterbi algorithm.

2.3 THE VITERBI ALGORITHM

2.3.1 Applications of Viterbi Decoding Algorithm

The Viterbi algorithm (VA) was proposed by Andrew J. Viterbi [35] in 1967 as a method of decoding convolutional codes. Recently Viterbi decoding algorithm, which was proved to yield maximum likelihood decisions [36], has become one of the most widely used forward error-correction techniques in digital communications. The Viterbi decoding algorithm is particularly desirable for efficient communication at very high data rates, where very low error rates are not required, or where large decoding delays are intolerable.

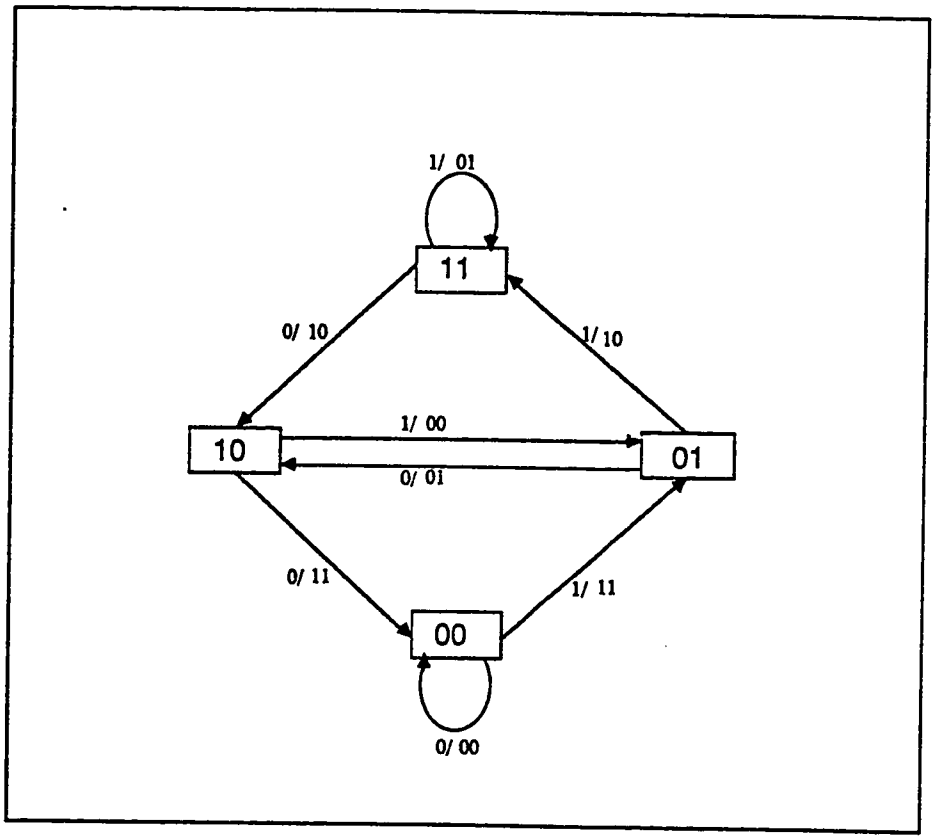


Figure 2.5 : State diagram for a constraint length of 3

This decoding algorithm has found numerous applications in communication systems. It has been shown [17] that Viterbi decoding is a practical method for improving the efficiency of satellite and space communications. This is due to the fact that the primary additive disturbance on space and satellite channels can usually be accurately modeled by Gaussian noise which is essentially independent from one bit time interval to the next. Two effective decoding algorithms for memoryless noise channels have been recommended, namely sequential and Viterbi decoding of convolutional codes [17]. The sequential decoding advantage tends to increase as lower probability of bit error are demanded, while Viterbi decoding is recommended for systems that are sensitive to bursts of errors [17]. The Viterbi decoding algorithm has been proved to provide performance superior to all other decoding schemes at high data rate, and with considerable robustness against varying channel parameters [11]. The Viterbi algorithm has also been applied to intersymbol interference problems, it has been found that on most channels intersymbol interference need not degrade the performance significantly [11]. The Viterbi algorithm has also found applications in areas other than digital communications. Text recognition, and pattern recognition are examples of these applications [11].

2.3.2 The Viterbi Algorithm:

The Viterbi decoding algorithm can be thought of as attempting to

find a path through the trellis structure which matches the received sequence as closely as possible, that is, to attempt to minimize the sequence error probability. Although VA does not guarantee that the actual bit error rate will be minimized, it is guaranteed that making the sequence error rate small will also make the bit error rate small for most cases. In fact, the bit error rate will be small for all but pathologically bad codes which are of no practical importance.

Finding the path which matches the received sequence as closely as possible can be done by calculating the Hamming distance between the trellis branches and the received sub-block. This Hamming distance can be used as a metric, and only the branch with the best (minimum) metric is retained and is called the survivor. The other branches are discarded. With this convention, a small branch metric represents a highly probable event, while larger metrics represents a less likely one. Therefore the Viterbi algorithm is optimum in the sense that it always finds the maximum likelihood path through the trellis. One problem may arise which is the possibility that in a given comparison between merging paths, the metrics are identical. Then we may simply flip a coin to choose one from them. For even if we preserved both of the equally valid contenders, further received symbols would effect both metrices in exactly the same way and thus not further influence our choice [36].

In its most general form, the Viterbi algorithm may be viewed as a

solution to the problem of maximum a posteriori probability estimation of the state sequence of a finite-state discrete-time Markov process observed in memoryless noise [11]. VA can also be thought of as finding the shortest route through a certain graph. In order to state VA in a formal way let's refer to the binary trellis diagram shown in Figure 2.4. As depicted in the figure, the state s_i at time k is one of a finite number X of states s, $1 < x < X$. Assume the state sequence is represented by a finite vector $S = (s_o, -----s_K)$. Define the transition ξ_k at time k as the pair of states (s_{k+1}, s_k):

$$\xi_k = (s_{k+1}, s_k)$$

Let $\hat{s}(s_k)$ indicates the survivor path corresponding to s_k. For any time there are X survivors in all, one for each s_k. At any time k, we need remember only X survivors $\hat{s}(s_k)$ and their lengths

$$\Gamma(s_k) = \lambda\{\hat{s}(s_k)\}$$

The formal statement of the algorithm is shown in Figure 2.6

2.3.3 Illustrative Example:

Consider the following situation, as an illustrative example of how the Viterbi algorithm corrects a specific error pattern. Assume that the encoder shown in Figure 2.2 is used to transmit all zero sequence. Assume also, that the received pattern is the sequence 01 00 00 10 00 00 00 00 -------. A series of incomplete trellis

Storage

k (time index)

$\hat{s}(s_k)$, $1 \le s_k \le X$ (Survivor terminating in s_k),

$\Gamma(s_k)$, $1 \le s_k \le X$ (Survivor length).

Initialization:

$k = 0$,

$\hat{s}(s_0) = s_0$, $\hat{s}(x)$=arbitrary, $x \neq s_0$,

$\Gamma(s_0) = 0$, $\Gamma(x) = \infty$ $x \neq s_0$,

Recursion: Compute

$$\Gamma(s_{k+1}, s_k) = \Gamma(s_k) + \lambda\{\xi_k(s_{k+1}, s_k)\}$$

for all $\xi_k = (s_{k+1}, s_k)$,

Find

$$\Gamma(s_{k+1}) = \min_{s_k} \Gamma(s_{k+1}, s_k)$$

for each s_{k+1}, store $\Gamma(s_{k+1})$ and the corresponding survivor $\hat{s}(k+1)$

Set k to k+1 and repeat until k=K

Figure 2.6: The Viterbi Algorithm

diagrams appear in Figure 2.7 that illustrate the status of the decoder after each received branch is processes. The numbers attached to each node indicate the Hamming distance, accumulated by the surviving path at that node. At level 3 each of the paths at level 2 is extended in two directions (since r = 2) making a total of eight paths. The metrics for the two paths entering each node are then compared and only the best path is retained as shown in Figure 2.7a. Repeating the same procedure at each level the resulting survival paths at levels 4,5,6 and 8 are shown in Figure 2.7. It is to be noted that all survivor paths tend ultimately, to lead back to a unique node and hence to a unique information symbol. In this example, if the decoding process is continued, then eventually only the all-zero path will survive. The depth at which this merging takes place is a random variable which depends upon the severity of the error pattern. In practical implementations one does not rely on this mechanism but rather establishes a fixed decoding depth.

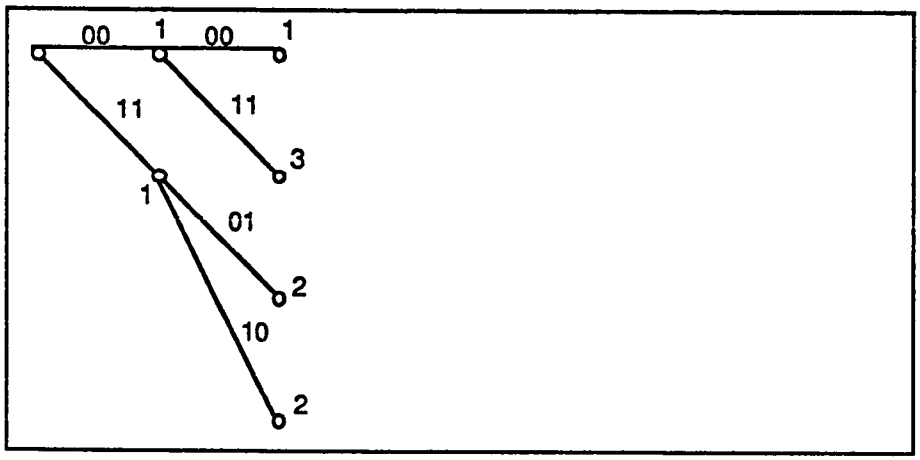


Figure 2.7 a: Status of the decoder and the survived paths at stage 2

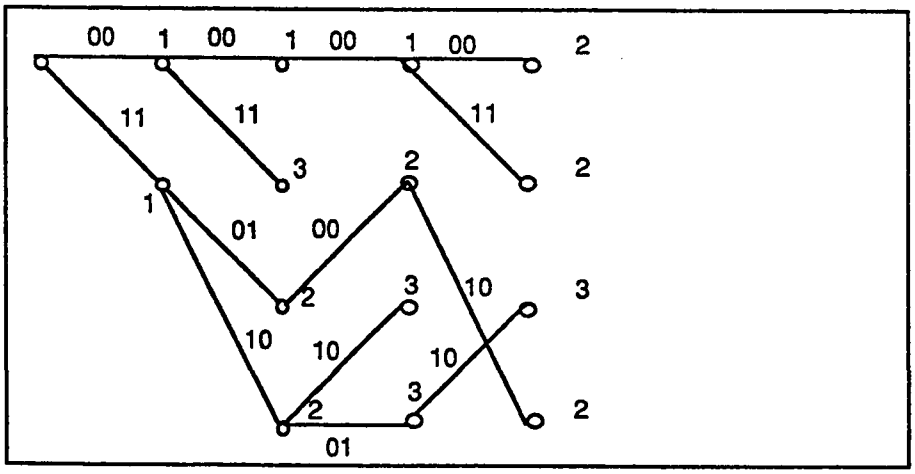


Figure 2.7 b: Status of the decoder and the survived paths at stage 4

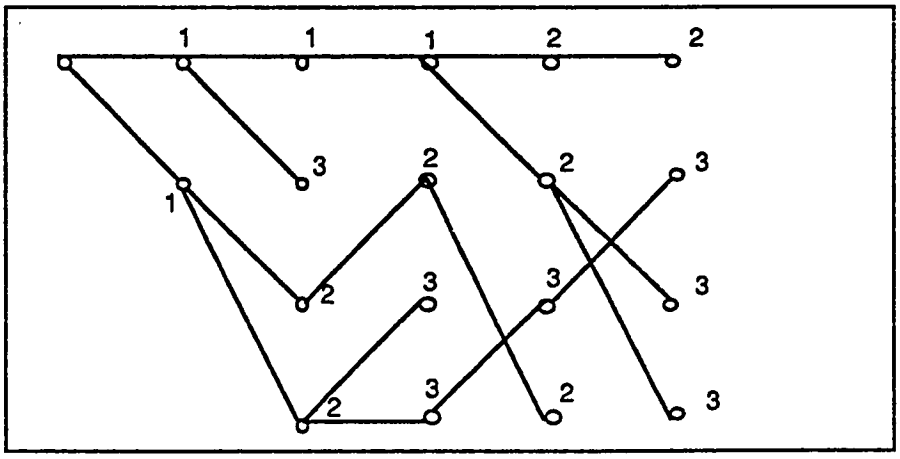


Figure 2.7 c: Status of the decoder and the survived paths at stage 5

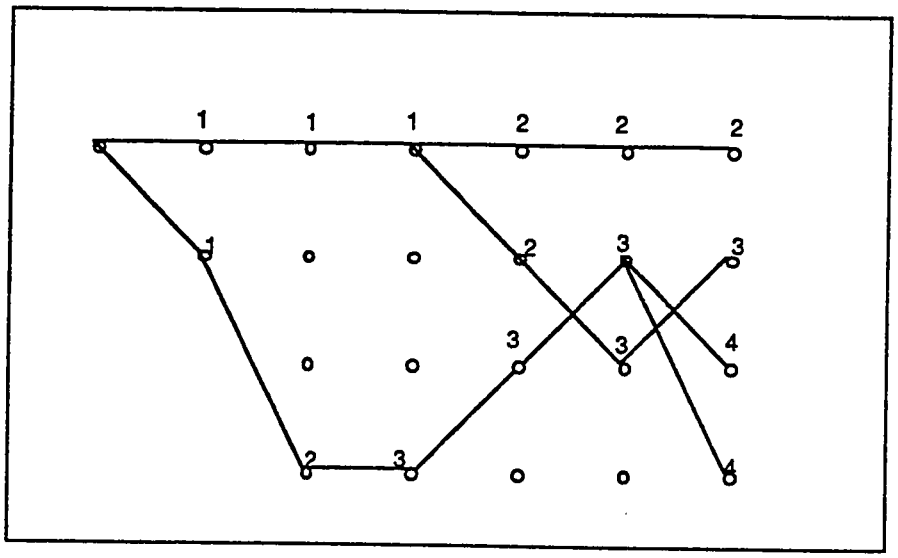
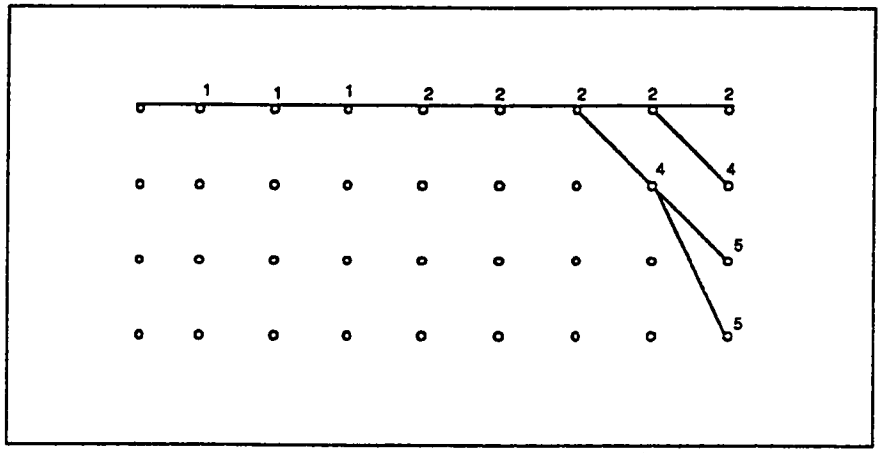


Figure 2.7 d : Status of the decoder and the survived paths at stage 6 of the trellis structre



. 2.7 e : Status of the decoder and the survived paths at stage 8

CHAPTER III

SYSTOLIC ARRAYS

3.1 INTRODUCTION

The tremendous progress in VLSI technology has increasingly encouraged the implementation of complex algorithms directly in hardware. The tendency of systolic algorithms to provide simpler, regular and modular layouts make them well-suited for VLSI implementation. Figure 3.1 depicts a Y-chart that shows the process of designing algorithmically specified VLSI digital systems [12].

As indicated in the Y-chart, a functional specification of the components is translated first into a structural representation and then into a geometrical description. The arrows in the figure represent design procedures that translate one phase of representation into another. Arrows drawn along each axis and pointed toward the origin indicates a top-down design procedure.

3.2 SYSTOLIC ARRAYS

3.2.1 Definitions and Concepts

The concept of "systolic arrays" was first introduced by H.T. Kung and C.E. Leiserson in 1978 [19]. Systolic arrays are a methodology for combining logic and memory to create powerful regular structures. Systolic arrays consist of a network of simple processors (cells) that circulate data in a regular fashion so that processor and communication resources can be fully utilized. Data and control flow at the array is usually simple and regular, so that communication among cells is local. Long distance or irregular communications are usually avoided. Every processor in the array rhythmically pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the system. One important feature of the systolic arrays is that the array uses extensive pipelining and multiprocessing so that a large proportion of the processors in the array are kept active which results in sustaining a high rate of computation flow.

The term "systolic", borrowed from physiologists, was chosen to draw an analogy with the human circulatory system, in which the heart sends and receives blood as a result of the frequent and rhythmic pumping of small amounts of blood through the arteries and veins. In this analogy, the heart corresponds to a source and destination of data, while the network of veins corresponds to the array of processors and

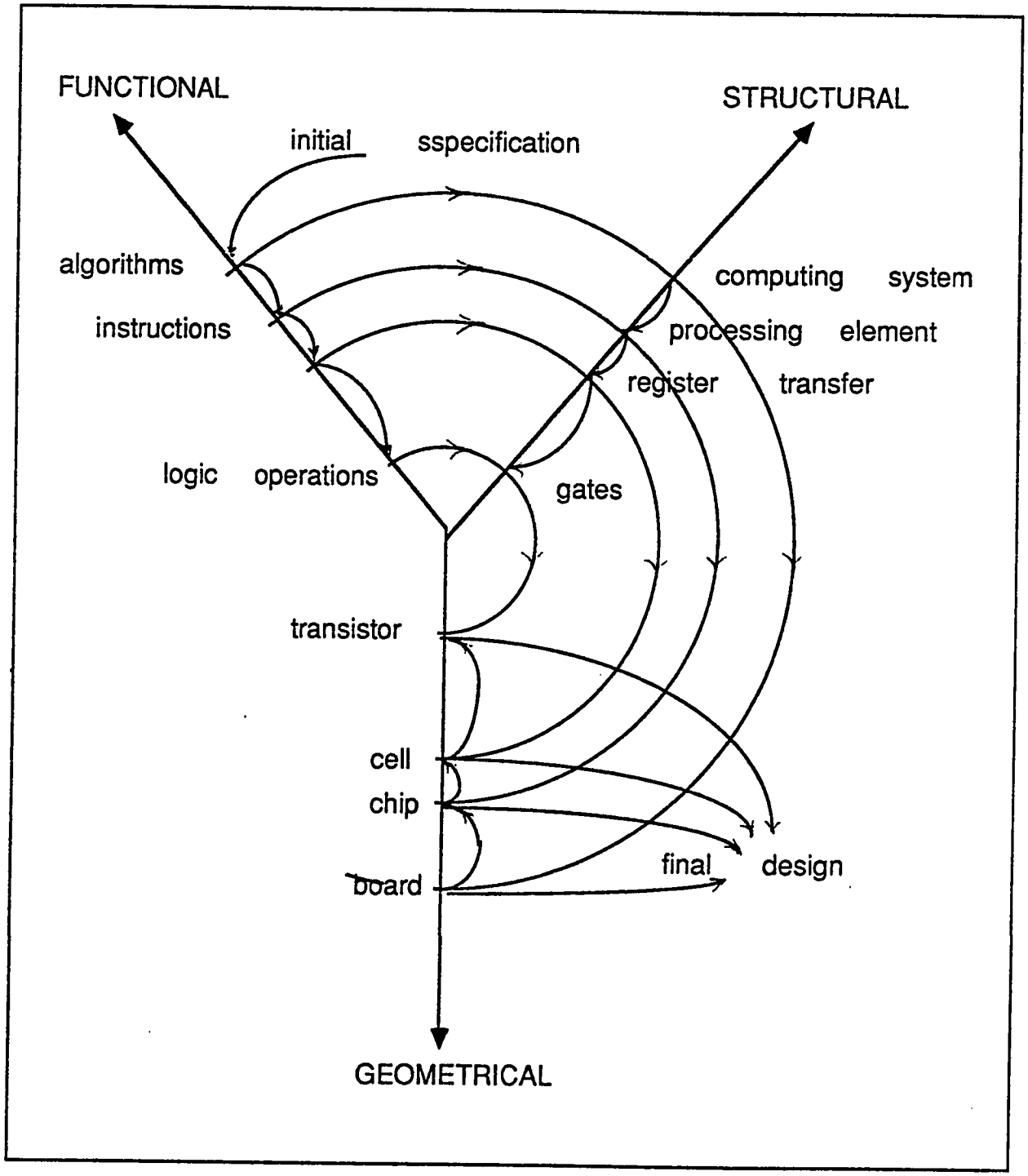


Figure 3.1 : Design of Digital Systems

links.

The fact that systolic arrays require synchronizing executing processors has made systolic system design quite difficult. However, there has been considerable progress in the development of techniques for simplifying systolic design [23]. Partitioning [27], finding recursive relations [20], and modeling using block diagrams [20] are examples of simplifying the design of systolic arrays.

An outline for a design methodology of systolic arrays is provided below:

3.2.2 Design Methodology

a) Choice of the Algorithm

Algorithms that are efficiently executed on systolic array processor are called systolic algorithms. Systolic algorithms should possess the properties of regularity, pipelining, multiprocessing and local communications. Systolic algorithms should also schedule computations in such away that a data item is not only used when it is input but also is reused as it moves through the pipelines in the array.

b) Design of Basic Processors:

The systolic algorithm is mapped into processors in which operations are performed and data is manipulated. The operation performed in each processor can range from a simple bit-wise operation, to word-level

multiplication and addition and even to execution of a complete program. The choice of granularity is determined by the application, or the technology, or both.

c) Timing and Synchronization:

In systolic arrays it is essential to "time" all operations and data transfer such that the data streams are synchronized and meet each others at the appropriate instants. An alternative to the design of a globally synchronous array is to use the so called wavefront arrays in which a self-timed system is achieved through the use of asynchronous handshaking mechanisms established between neighboring processors.

d) Complete Array Design:

Identical processors are grouped in such a way that processors, which communicate directly with each other, are put together in an efficient way to maintain regular network. The resulting array is used to perform the required task.

e) Scaling and Simulation:

It is essential to be sure of functional correctness. Tests should be made on the systolic system by checking the status and accumulation of the results at each clock cycle, starting from the time the data stream enters the array until the time a sufficient output is generated. A simulation can also be used to check for functional correctness if a

proper software is available. A register transfer language (RTL) like UAHPL can be used to simulate large kinds of systolic algorithms. In some cases, scaling is performed to increase the throughput of the array or to solve time synchronization problems.

CHAPTER IV

PROPOSED SYSTOLIC ALGORITHM FOR A VITERBI DECODER AND ITS SIMULATION

4.1 INTRODUCTION

Tremendous progress in MOS technology has made custom digital VLSI very attractive. In this chapter the application of systolic architectures to the design of Viterbi decoders is considered. A new proposed systolic algorithm for designing Viterbi decoders, using the design methodology discussed in chapter 3, is provided in section 4.2. The goal is to meet the speed requirement and to optimize the silicon area, power consumption and design time. Section 4.3 provides a UAHPL simulation for the basic cells of Viterbi decoders. In section 4.4 the output of simulation for a Viterbi decoder is discussed in detail. The decoder output is discussed in section 4.5.

4.2 THE PROPOSED SYSTOLIC ALGORITHM

4.2.1 Systolic Algorithm Philosophy

The nature of Viterbi decoding make it feasible to describe it by a systolic algorithm. There are four basic operations the systolic algorithm ought to perform in order to successfully decode convolutional codes using Viterbi algorithm. These basic operations are as follows:

1) Branch metric computation: This is the basic computation element of the algorithm. With every received sub-block of input sequence, a new set of metric values, with respect to all of the trellis branches, have to be calculated.

2) Path metric updating: The branch metrics are added to the previously stored path metrics. Then, for each node a comparison is made among all the merging paths (For binary trellis there are only two merging paths for each node), and the smallest path is selected and stored as the new path metric.

3) Information sequence updating: Once the survived paths are chosen for all nodes, the history for each path is updated and stored.

4) Decoder output: The survivor path which contains the information sequence is selected after repeating all the operations mentioned above starting from the discrete time k=j to k=j+l, where j is the integer and l is the search length. The information bits on the

selected survivor are chosen as the decoded bits.

In our proposed systolic algorithm the first three operations are distributed between two blocks of cells. Each block consists of identical processors operating in a highly parallel scheme. The first block consists of a column of processors which perform branch metric computation and accordingly decides on the survived branches. The second block consists of a matrix of simpler processors which perform both path and information sequence updating. The decoder output, which is the last operation in the four operations listed above, is carried out using simple Boolean functions.

The systolic algorithm [31] can be stated formally as follows:

Procedure PROC($S_v S_{v-1} \ldots S_2 S_1$)

Global: INPUT, OUT, FLG_1, FLG_2

Local : $h_1, h_2, \Sigma_1, \Sigma_2, X, Y, R_1, R_2, \alpha, \beta$

Initialization of registers

if $S_v S_{v-1} \ldots S_1 = 0..00$ **or** $S_v S_{v-1} \ldots S_1 = 0..01$ **then**

$R_1 = 0, \quad R_2 = \infty$

else

$R_1 = \infty, \quad R_2 = \infty$

endif

Initialize α and β

1. $X = \alpha\{1..n\}@INPUT\{1..n\}$,

$Y = \beta\{1..n\}@INPUT\{1..n\}$,

$h_1=0, \quad h_2=0,$

for k = 1 to n Step1 do

if$(X_k=1)$ $h_1=h_1+1,$

if$(Y_k=1)$ $h_2=h_2+1,$

end[for]

$\Sigma_1=h_1+R_1,$

$\Sigma_2=h_2+R_2,$

$OUT(S_vS_{v-1}....S_2S_1)=\min\{\Sigma_1,\Sigma_2\},$

$R_1=OUT(S_{v-1}....S_1S_v)$

$R_2=OUT(S_{v-1}....S_1\overline{S_v})$

if $(\Sigma_1 \le \Sigma_2)$ **then,**

$FLG_1(S_vS_{v-1}....S_1)=1$

$FLG_2(S_vS_{v-1}....S_1)=0$

else

$FLG_1(S_vS_{v-1}....S_1)=0$

$FLG_2(S_vS_{v-1}....S_1)=1$

endif

Go to 1

Procedure $P(S_v S_{v-1} \ldots\ldots S_2 S_1)$

Global: $PATH, FLG_1, FLG_2$

Local : $FLAG_1, FLAG_2$

1. **if** $PATH(S_{v-1} \ldots\ldots S_2 S_1 S_v)=0$ and

$PATH(S_{v-1} \ldots\ldots S_2 S_1 \overline{S_v})=0$ **then**

$FLAG_1(S_v S_{v-1} \ldots\ldots S_1)=0$

$FLAG_2(S_v S_{v-1} \ldots\ldots S_1)=0$

else

$FLAG_1(S_v S_{v-1} \ldots S_1)=FLG_1(S_v S_{v-1} \ldots S_1)$,

$FLAG_2(S_v S_{v-1} \ldots S_1)=FLG_2(S_v S_{v-1} \ldots S_1)$

endif

Go to 1

In the above algorithm, R_1 and R_2 indicate the content of first and second registers in each processor respectively. Σ_1 and Σ_2 represent the first and second adders in each processor. h_1 and h_2 indicate the Hamming distance between branches (α or β) and the received sub-block (INPUT). $S_v S_{v-1} \ldots\ldots\ldots S_2 S_1$, is a binary string identification for each processor (row wise). It is to be noted that two processors PROCK and PROCK' are linked via a *shuffle* if PROCK' is a left or right cycle shift of PROCK (i.e., if PROCK = $S_v S_{v-1} \ldots\ldots\ldots S_2 S_1$,

then PROCK' = S_{v-1}..... S_2 S_1 S_v, or PROCK' = S_1 S_v S_{v-1}..... S_2). Two processors PROCK and PROCK' are linked via an ***exchange*** if PROCK and PROCK' differ only in the first bit (i.e., if PROCK = S_v S_{v-1}..... S_2 S_1 then PROCK' = S_v S_{v-1}..... S_2 $\overline{S_1}$ where $\overline{S_1}$ is the logical complement of S_1). Two processors are linked via an ***exchange-shuffle*** if the two operations are carried out simultaneously.

It is to be noted that the above algorithm applies only to binary trellis, nevertheless, extending the systolic algorithm to higher alphabetic levels is straight forward (see chapter 6).

4.2.2 Architecture of the Two Blocks

The systolic algorithm mentioned in the previous section (4.2.1) can easily be mapped into hardware implementation. A block diagram of the first and second block is shown in Figure 4.1. The architecture of the two blocks can be described as follows:

a) Architecture of first block

As shown in Figure 4.2 the first block consists of one column of r^{m-k} identical processors. Each processor consists of r registers, r adders, and r flags, where r indicates the number of alphabetic levels used in the trellis structure. From now on, we will focus on the binary trellis

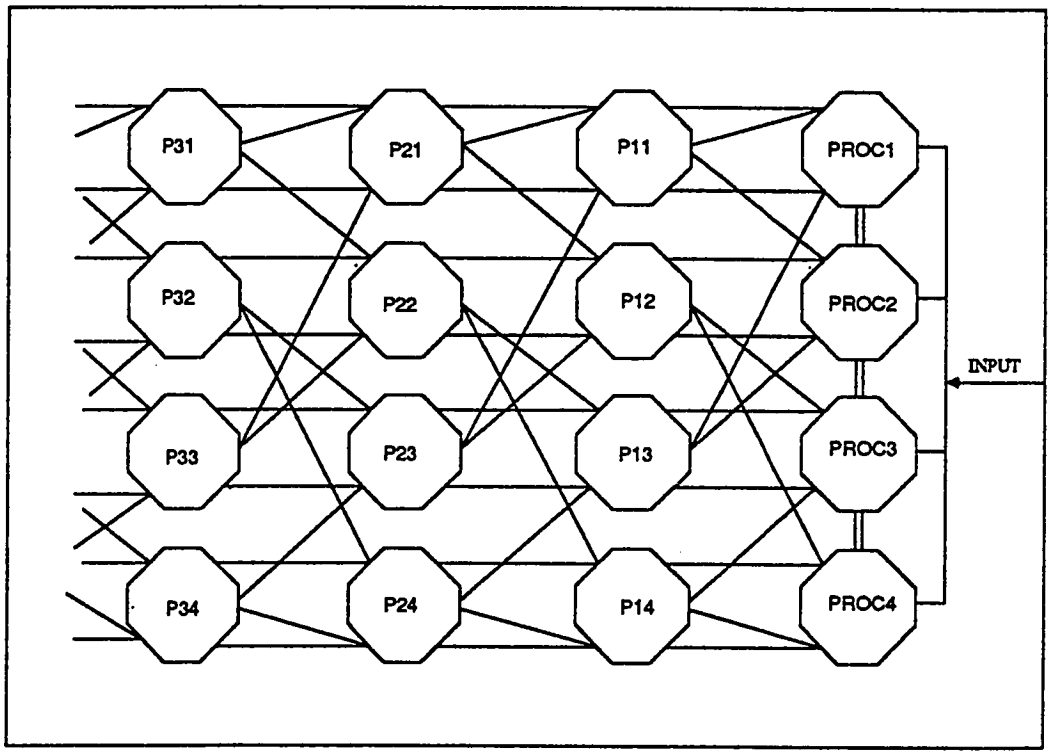


Figure 4.1: Architecture design of the systolic algorithm for m=3

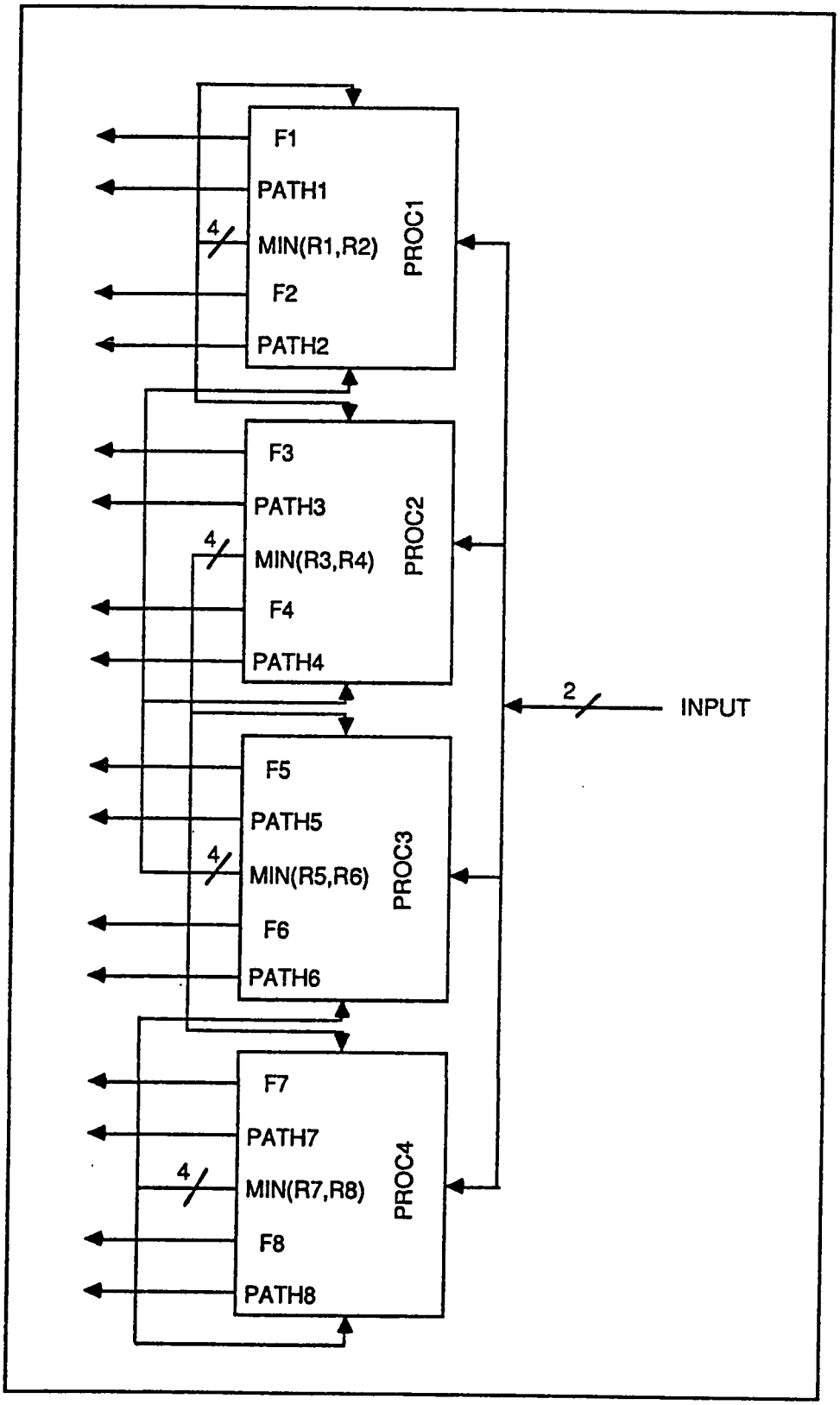


Figure 4.2 a : Interconnection between processors of the first block

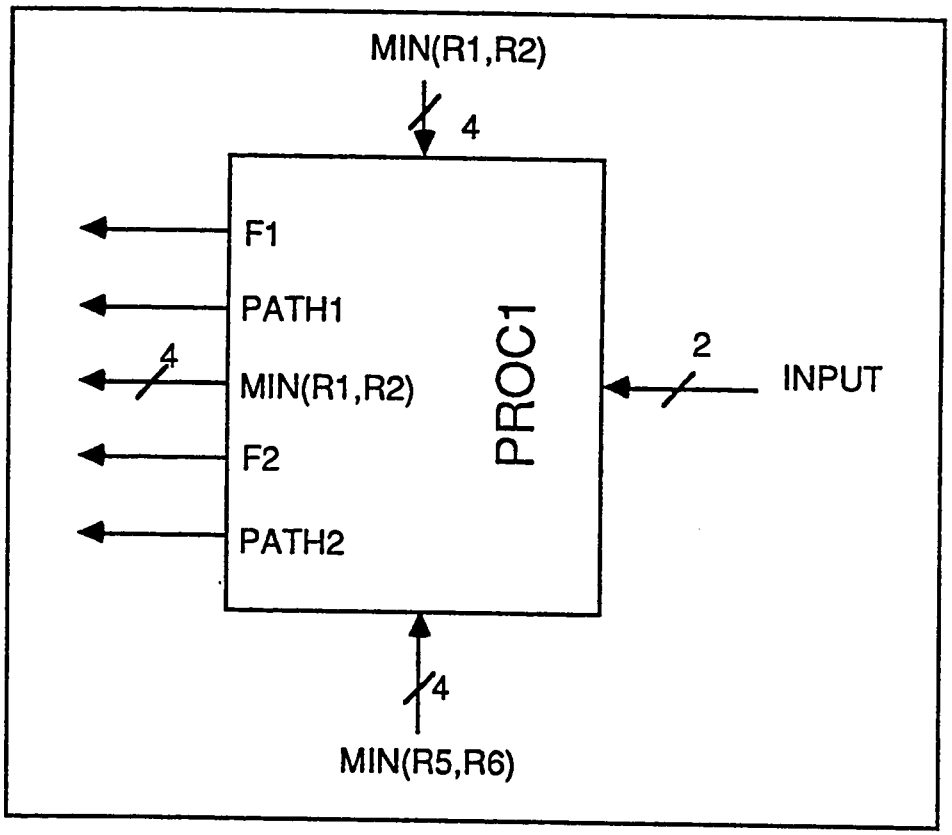


Figure 4.2 b : a first block processor

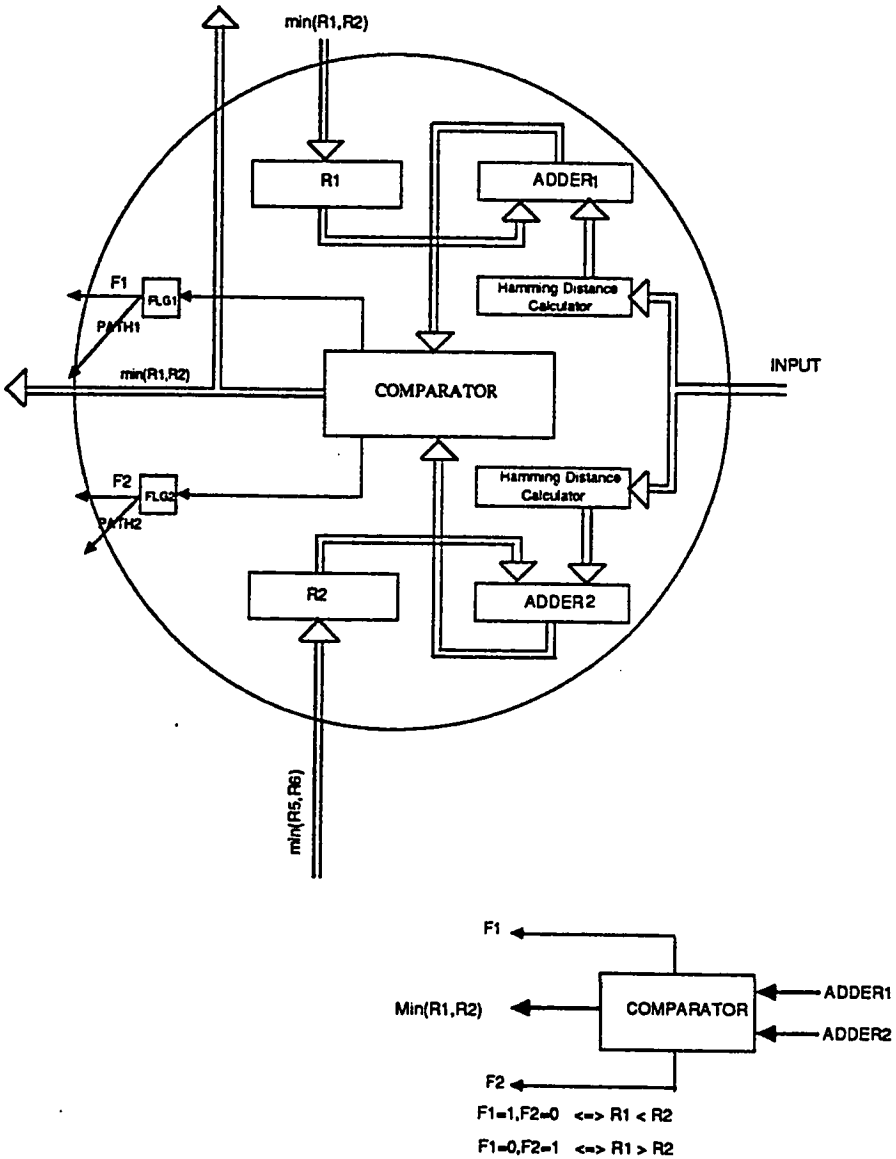


Figure 4.2 c
Architecture of first block

where r=2. The analysis can easily be extended to higher levels of alphabets. Branch metric computation is accomplished in each processor by adding the Hamming distance to the path metric of previous stage which is stored in one of the two registers in the processor. The branch which corresponds to the register that has the minimum distance is the one which is selected to be the survived branch and whose corresponding flag is set to 1, while the other flag remains 0. It is to be noted that in case the two Hamming distances have the same metric, then any of the two paths could be considered the survived path and its corresponding flag is set to 1. In this design if equal metrics take place, then the upper flag of the processor of concern is set to 1, while the other flag is set to 0. This operation is repeated in every clock pulse, and the contents of the registers and the flags are updated accordingly. It is to be noted that in every clock pulse a comparison between the contents of registers of every processor is made, the minimum of the two is transfered to two other registers. These two registers are: i) register R_1 of a processor whose binary address representation is defined as a shuffle operation of binary address representation of the source processor. ii) register R_2 of a processor whose binary address representation is defined as a shuffle-exchange operation of binary address representation of the source processor. This process can be mapped in the systolic algorithm as choosing the path metric for each node of the trellis structure (see Figure 4.1). The contents of the flags of previous clock pulse are

pumped to the next block as shown in Figure 4.1 (also see algorithm).

Overflow of the finite length register is prevented by subtracting a constant from all the registers provided that the contents of all registers are greater than zero. The value of the constant to be subtracted depends on the rate of the encoding; for example, in the case of rate 1/2 code, subtracting a one in every clock pulse prevents registers from overflow except for the case of a very severe error pattern which is however, non-correctable even if the registers did not overflow. It may be shown that with a 4-bit register, the probability of register overflow, under normal channel conditions, ie. bit error rate less than 10^{-3}, is negligible.

b) Architecture of second block

As mentioned earlier the second block consists of a matrix of i*j simple processors as shown in Figure 4.3, where i equals the number of states in the trellis structure ($i= r^{m-k}$), and j equals the number of stages of trellis structure before which the decoder starts to transmit the decoded output. The greater the value of j, the greater is the capability of the decoder to give a correct output. Each processor has r flags (r=2 for the binary trellis case) of which one flag is set to 1, and the other flags remain 0. Transfer of contents of processors in column l of the trellis structure, to those in column l+1 takes place at every clock pulse, and the history of survived paths is stored.

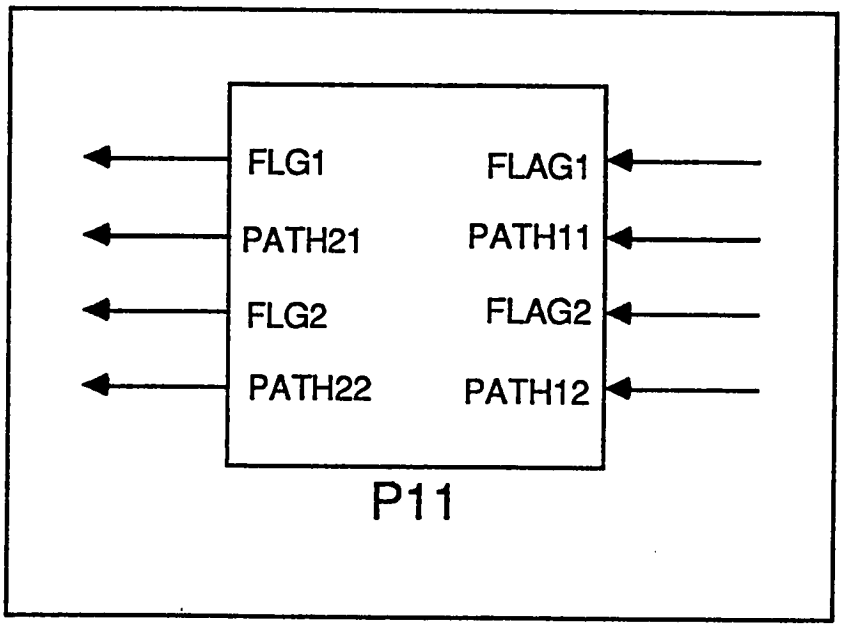


Figure 4.3 a: Design of second block processor

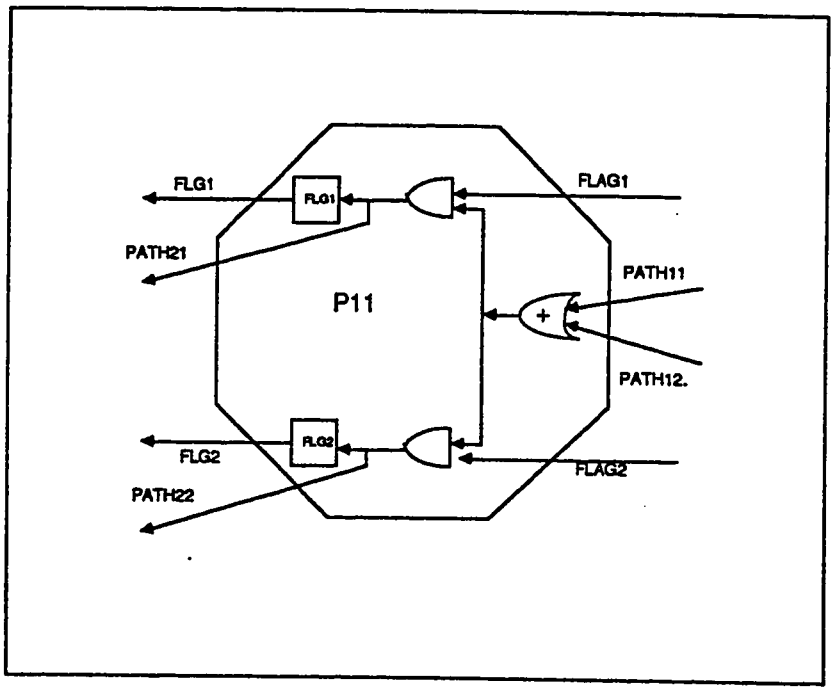


Figure 4.3b: Design of second block processor

Contents of flags in processors of the second block indicate the survived paths of the trellis. The upper flag of each processor represents the status of the upper branch for the corresponding node in the trellis structure, while the lower flag represents the lower branch. A logic 1 in the upper flag and 0 in the lower one indicates that the upper branch is the survived branch and vice versa. Thus by scanning the contents of the two flags from right to left in each processor of the second block it is possible to determine a survived path. In case of merging of paths at a certain node, as a result of decoding process, the unsurvived paths are erased by clearing all the corresponding flags of the unsurvived paths. This is done in each clock pulse. One feature of the above architecture that must be emphasized is: since the processors in the second column are all identical and simple, the number of columns can conveniently be increased to enhance decoding capability.

4.3 RTL MODEL AND SIMULATION

In order to check for functional correctness of the systolic algorithm, the algorithm has been modeled in a Universal AHPL and has been tested using a functional level simulator. "AHPL is a register transfer level language which has been used for more than a decade for teaching basic concepts of computer design [32]. The language is simple, elegant and powerful and is sufficiently general to accommodate

the description of both simple and highly complex digital systems. Such a range spans simple combinational circuits to complex parallel processors and data flow machines. Inspite of this, AHPL lacked certain constructs and features, which have little pedagogical significance, but are necessary for efficient realization and testing of digital systems. UAHPL (Universal AHPL) has been designed after reviewing the requirement of a wide spectrum of design and test activities in various environments. The new language gives more freedom in the choice of register types, clocking options and bus types. It accommodates pass transistors, wired-OR gates, temporary registers and other elements, which are useful for an efficient realization of a circuit in MOS technology [32]."

A multi-stage compilation process has been used in UAHPL in order to support a wide spectrum of design and test activities. " The first stage accepts UAHPL circuit description, performs syntax analysis and semantics checking and decomposes the source text into a tabular representation of the circuit [32]". Stage-2 compiler constructs the complete circuit in terms of memory elements, buses, gates, etc, and their interconnection. Stage-3A processor uses stage-1 tables and simulator directives to perform functional simulation at the register transfer level and to produce simulation output. " The multi-stage approach has significant advantages. The designer of a stage-3 processor need not worry about the task of syntax analysis, semantic checking and other such compilation issues. The design process starts

with a much easier to manage tabular and linked list representation of the circuit. Thus, an application-specific stage-3 can be written in a much shorter time. Also, since stage-1 and stage-2 are free from application specific details, they are more 'modular and are easily modifiable". Figure 4.4 depicts a block diagram of the UAHPL DA system.

The systolic algorithm has been modeled in an RTL and has been tested using a functional level simulator (Stage-3A of UAHPL DA). The output of the simulation ascertained the correctness of the algorithm. Figure 4.5 shows the RTL model of processor of first block, for a 1/2 rate convolutional code. Figure 4.6 depicts the RTL model of a processor used in the matrix of block 2. An illustrative example showing the output of simulation for a systolic system is provided in the next section.

4.4 SIMULATION OUTPUT FOR A VITERBI DECODER

A Viterbi decoder has been simulated using UAHPL. The Viterbi decoder simulated is a 1/2 rate convolutional decoder whose constraint length m=3 and whose trellis structure is identical to the one shown in Figure 2.3. The block diagram of the Viterbi decoder is depicted in Figure 4.1. An RTL model of the decoder is listed in Appendix A. Note that an overflow in the registers has been avoided by subtracting a one

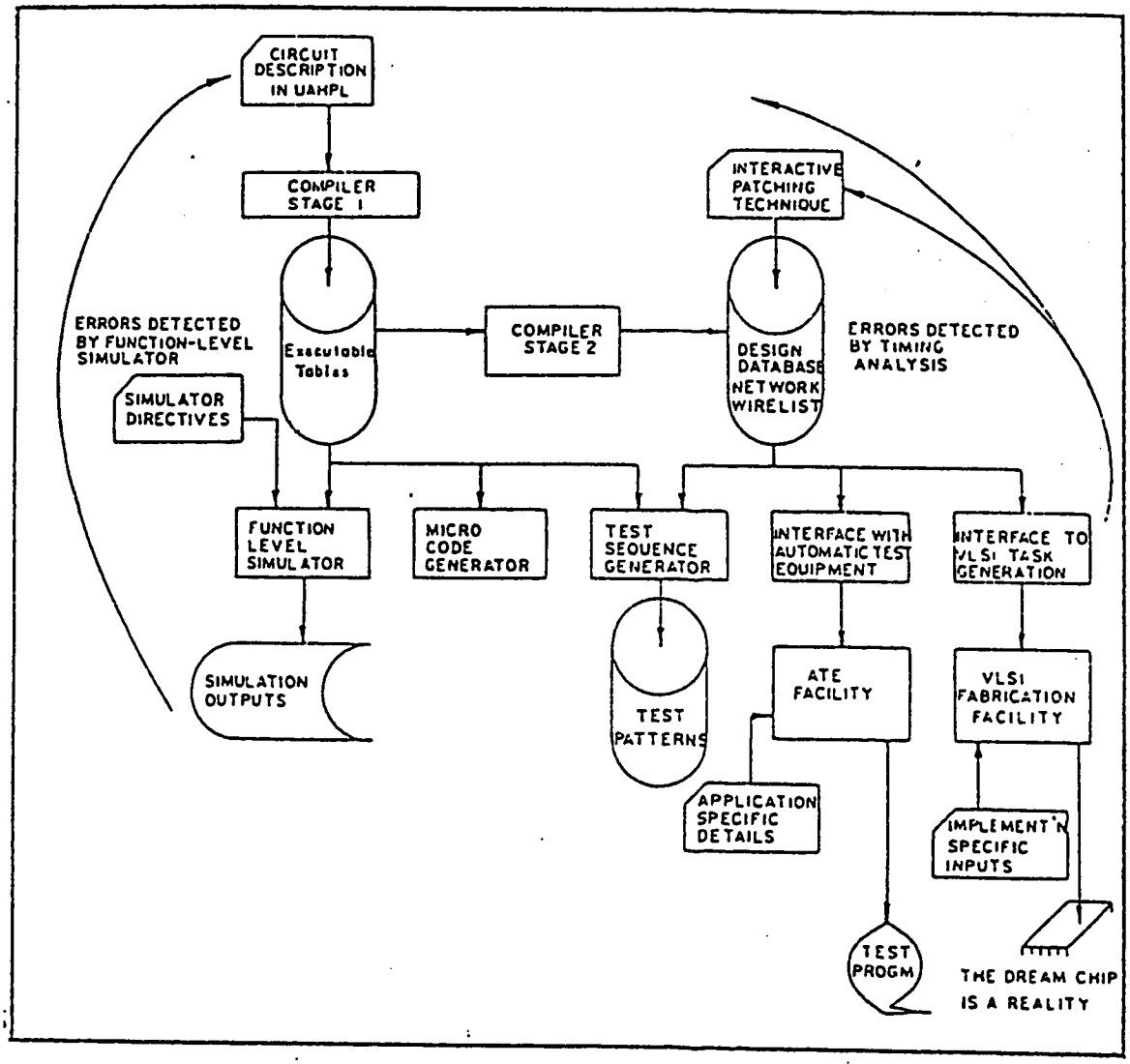


Figure 4.4: Block diagram of a UAHPL based DA system

```
=======================================
 RTL Model of processor of FIRST block
=======================================
MODULE      :PROC1.
  MEMORY    :REG1[4]; REG2[4].
  MEMORY    :FLG1; FLG2.
  EXINPUT   :IN[2].
  EXINPUT   :CLOCK; RESET.
  BUSES     :G1; G2; G3.
  BUSES     :C[3]; D[3]; IMIN12[4].
  BUSES     :PREG1[4]; PREG2[4].
  BUSES     :M1[4]; M2[4].
  EXBUSES   :MIN12[4]; MIN56[4].
  EXBUSES   :F11; F12.
  EXBUSES   :PATH11; PATH15.
  CLUNITS   :COM[3]; ADD[4].
BODY
SEQUENCE : CLOCK.

    ====================
     Initialization step
    ====================

1   PREG1 = \0,0,0,0\;
    FLG1 <= \1\;
    REG1 <= ADD(2$0,G1,G2;PREG1).

 ================================
  Main function of the processor
 ================================

2   C     = COM(REG1; REG2);
    IMIN12= (C[2]&REG1 + ¬C[2]&REG2);
    PREG1 = MIN12;
    PREG2 = MIN56;
    M1    = ADD(2$0,G1,G2;PREG1);
    M2    = ADD(2$0,G3,G2;PREG2);
    D     = COM(M1;M2);
    FLG1 <= D[2];
    FLG2 <= ¬D[2];
    REG1 <= M1;
    REG2 <= M2;
    => 2.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
                                    cont.
```

```
    MIN12 = IMIN12;
    F11   = FLG1;
    F12   = FLG2;
    PATH11= FLG1;
    PATH15= FLG2;

"Hamming Distance Calculations"

    G1 = IN[0] & IN[1];
    G2 = IN[0] @ IN[1];
    G3 = ¬IN0] & ¬IN1].
END.
```

Figure 4.5

```
=======================================
 RTL Model of processor of SECOND block
=======================================
MODULE     : P11.
  MEMORY   : FLG11;FLG12.
  EXINPUT  : CLOCK;RESET.
  BUSES    : UPFLG1;UPFLG2.
  EXBUSES  : F11;F12;F21;F22.
  EXBUSES  : PATH11;PATH12.
  EXBUSES  : PATH21;PATH25.
BODY
SEQUENCE : CLOCK.
1    UPFLG1=F11&(PATH11+PATH12);
     UPFLG2=F12&(PATH11+PATH12);
     FLG11 <= UPFLG1;
     FLG12 <= UPFLG2;
     => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
   PATH21 = UPFLG1;
   PATH25 = UPFLG2;
   F21  = FLG11;
   F22  = FLG12.
END.
```

Figure 4.6

from all the registers provided that the contents of all registers are greater than zero.

The decoding capability has been tested using different encoded information sequences after passing through a channel where some errors are added. Simulation outputs assure the correctness of the decoder design.

As an illustrative example of how the decoder corrects a specific error pattern, Let us consider again the of the illustrative example of section 2.3.3, where all zero sequence is transmitted and the received pattern is the sequence 01 00 00 10 00 00 ---------. The received sequence has been fed to the decoder (through the COMSEC of UAHPL). The output of the function level simulator (Stage-3A of UAHPL) matches the output shown in section 2.3.3. Contents of different processors at different stages are shown in Table 4.1. The status of the decoder and the survived paths at each stage can be derived from Table 4.1. For example, the survived paths shown in Figure 2.7d can be obtained from Table 4.1 by noting the contents of the flags of the second block. In the Figure 2.7 each node in column l may have a branch coming from nodes in column l+1 from the left. As mentioned earlier branches entering the nodes may be classified as an upper branch and a lower branch. In case a branch exists, if a branch of a path entering a node in column l is an upper branch then the contents of flags Fi1 and Fi2 are 1 and 0 respectively, else they are 0

and 1. If no survived branch exists then both flags are 0. Scanning out the contents of flags in Table 4.1 will lead to the same survived paths shown in Figure 2.7(a,b,c,d,e) for different clock pulses. This shows that the outputs of the simulator are identical with the theoretical results.

4.5 DECODER OUTPUT

The output of the decoder is a function of the last column of the second block processors. If the decoding process was successful, in the sense that all the survivor paths led to a unique node and hence to a unique information symbol, then there is only one survivor path at the last column of the second block. The output of the decoder is then, simply that survived path which contains the information sequence bits. On the other hand if it so happened that there are more than one survived path then the path with smallest metric is the one which has the lowest probability of error.

The technique adopted in this research is to choose the upper most path to be the survived path and to discard the others. This is achieved by checking the contents of the flags of the upper most processor of last column as tabulated in Table 4.2. If the contents of flags of the upper most processor are both zero then the contents of flags of next processor (column wise) determine the output of the

decoder and the procedure shown in Table 4.2 is repeated.

The implementation of the decoder output is simple combinational circuit. The Boolean expression of the decoder output is:

$$F = \overline{A_0}A_1 + \overline{\overline{A_0}A_1\overline{B_0}B_1} + \overline{\overline{A_0}A_1\overline{B_0}B_1\overline{C_0}C_1} + \overline{\overline{A_0}A_1\overline{B_0}B_1\overline{C_0}C_1\overline{D_0}D_1} + \ldots\ldots$$

Where A_0 represents the first flag of the first processor, A_1 represents the second flag of the first processor, B_0 represents the first flag of the second processor (column-wise), B_1 represents the second flag of the second processor and so on.

Clck	PROC#	I	HDC1	HDC2	ADD1	ADD2	MIN	FLG1	FLG2	F11	F12	F21	F22	F31	F32	F41	F42	F51	F52
1	1	01	00	11	01	VL	01	1	0	0	0	0	0	0	0	0	0	0	0
1	2	01	11	00	01	VL	01	1	0	0	0	0	0	0	0	0	0	0	0
1	3	01	01	10	VL	VL	VL	0	0	0	0	0	0	0	0	0	0	0	0
1	4	01	10	01	VL	VL	VL	0	0	0	0	0	0	0	0	0	0	0	0
2	1	00	00	11	01	VL	01	1	0	1	0	0	0	0	0	0	0	0	0
2	2	00	11	00	03	VL	03	1	0	1	0	0	0	0	0	0	0	0	0
2	3	00	01	10	02	VL	02	1	0	0	0	0	0	0	0	0	0	0	0
2	4	00	10	01	02	VL	02	1	0	0	0	0	0	0	0	0	0	0	0
3	1	00	00	11	01	04	01	1	0	1	0	1	0	0	0	0	0	0	0
3	2	00	11	00	03	02	02	0	1	1	0	1	0	0	0	0	0	0	0
3	3	00	01	10	04	03	03	0	1	1	0	0	0	0	0	0	0	0	0
3	4	00	10	01	04	03	03	0	1	1	0	0	0	0	0	0	0	0	0
4	1	00	00	11	02	05	02	1	0	1	0	1	0	1	0	0	0	0	0
4	2	00	11	00	02	04	02	1	0	0	1	0	0	1	0	0	0	0	0
4	3	00	01	10	04	03	03	0	1	0	1	1	0	0	0	0	0	0	0
4	4	00	10	01	02	05	02	1	0	0	1	1	0	0	0	0	0	0	0
5	1	00	00	11	02	05	02	1	0	1	0	1	0	1	0	1	0	0	0
5	2	00	11	00	04	03	03	0	1	1	0	0	1	0	0	1	0	0	0
5	3	00	01	10	03	03	03	1	0	0	1	0	0	1	0	0	0	0	0
5	4	00	10	01	03	03	03	1	0	1	0	0	1	1	0	0	0	0	0
6	1	00	00	11	02	05	02	1	0	1	0	1	0	1	0	1	0	1	0
6	2	00	11	00	04	03	03	0	1	0	1	1	0	0	0	0	0	1	0
6	3	00	01	10	04	04	04	1	0	1	0	0	1	0	0	0	0	0	0
6	4	00	10	01	04	04	04	1	0	1	0	0	0	0	1	1	0	0	0

VL: Very Large number HDC: Hamming Distance Calculator
F11: FLG11,F12: FLG12,.....etc.

Table 4.1:Output obtained from RTL simulation for different clock pulses

First Flag	Second Flag	Decoder Output
1	0	0
0	1	1
0	0	Check next processor
1	1	Never occur

Table 4.2: Procedure for detertmining the decoder output

CHAPTER V

CMOS VLSI DESIGN OF BASIC CELLS

5.1 INTRODUCTION

Low power consumption and small semiconductor area tend to favor CMOS VLSI design for Viterbi decoders over other kinds of VLSI design. In chapter 4 the systolic architecture for a Viterbi decoder is obtained. The systolic architecture was divided into two basic kinds of processors. The operation of each processor and the communication among processors were discussed.

In this chapter, the detailed designs of different cells of 1/2 rate convolutional decoder are developed. A SPICE simulation for each cell is obtained. The floor plan of the CMOS VLSI design and the VLSI layouts is accomplished for each cell. Before proceeding in the details of the VLSI design, let's review the general concept of two-phase clocking scheme which has been adopted in the design.

5.2 TWO-PHASE CLOCKING SCHEME

As shown in Figure 5.1, two-phase clocking scheme provides two high non-overlapping signals from a one period duration of the normal clock pulse. Two-phase clocking scheme offers the following advantages:

a) It allows data-division multiplexing i.e., different devices can share the same input and output paths if they are clocked by different phases (φ_1 and φ_2). This feature facilitates data manipulation of complex digital system and reduces hardware required.

b) It allows ordered-operations to be executed in the same clock pulse. This feature can be achieved by clocking the first device at phase I (φ_1), while clocking the second one at phase II (φ_2). This results in enhancing the overall system speed. In fact, this feature is used in the MOS design of first processor as will be shown later in this chapter.

c) It allows different devices to be clocked independently during the same clock cycle.

The circuit needed to derive phase I (φ_1) and phase II (φ_2) is simple and is depicted in Figure 5.2.

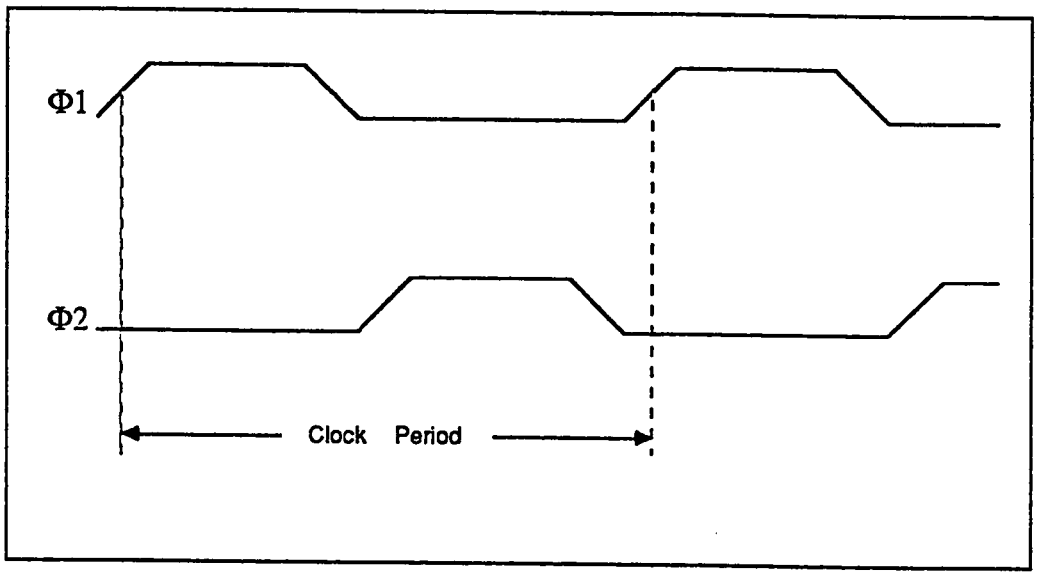


Figure 5.1 : Two Phase Clocking Scheme

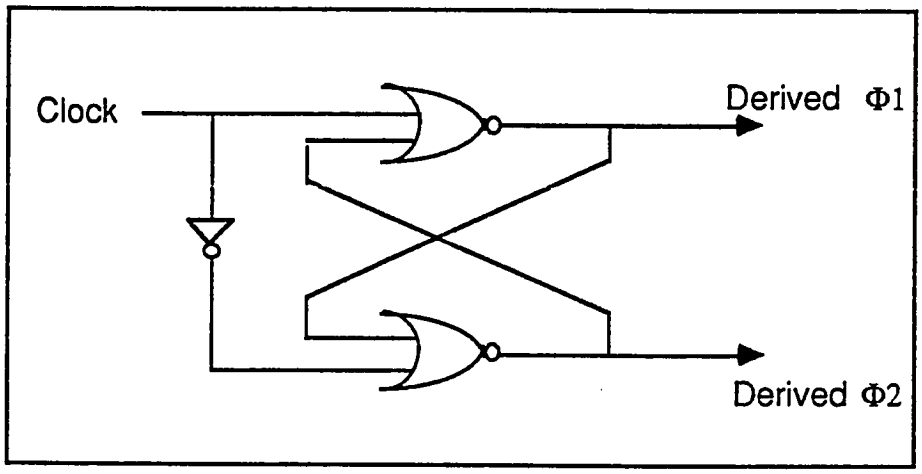


Figure 5.2 : Circuit to Derive Φ1 and Φ2

5.3 DESIGN OF FIRST BLOCK

As highlighted in previous chapter, first block for the 1/2 rate convolutional code consists of the following basic cells:

a) Four-bit full adders

b) Four-bit magnitude comparator

c) Four-bit registers (latch)

d) Hamming distance calculators (HDC)

e) Flags

The design of each cell is explained in detail below.

5.3.1 Four-Bit Full Adder Design

Logic diagram of a full adder is shown in Figure 5.3 [8]. The sum S and the output carry C_n of the full adder is given by

$$S = A \oplus B \oplus C_{in} \qquad \text{-------- (5.1)}$$

$$C_o = AB + (A \oplus B)C_{in} \qquad \text{-------- (5.2)}$$

where A and B are the adder inputs and C_{in} is the input carry. Note that @ indicates the logical EXOR operation. C_o can also be expressed in Boolean logic as $C_o = (\overline{A \oplus B})B + (A \oplus B)C_{in}$ for explicit mapping to the circuit shown in Figure 5.3. The four bit adder is composed of cascading four cells of the full adder. Random logic approach is used

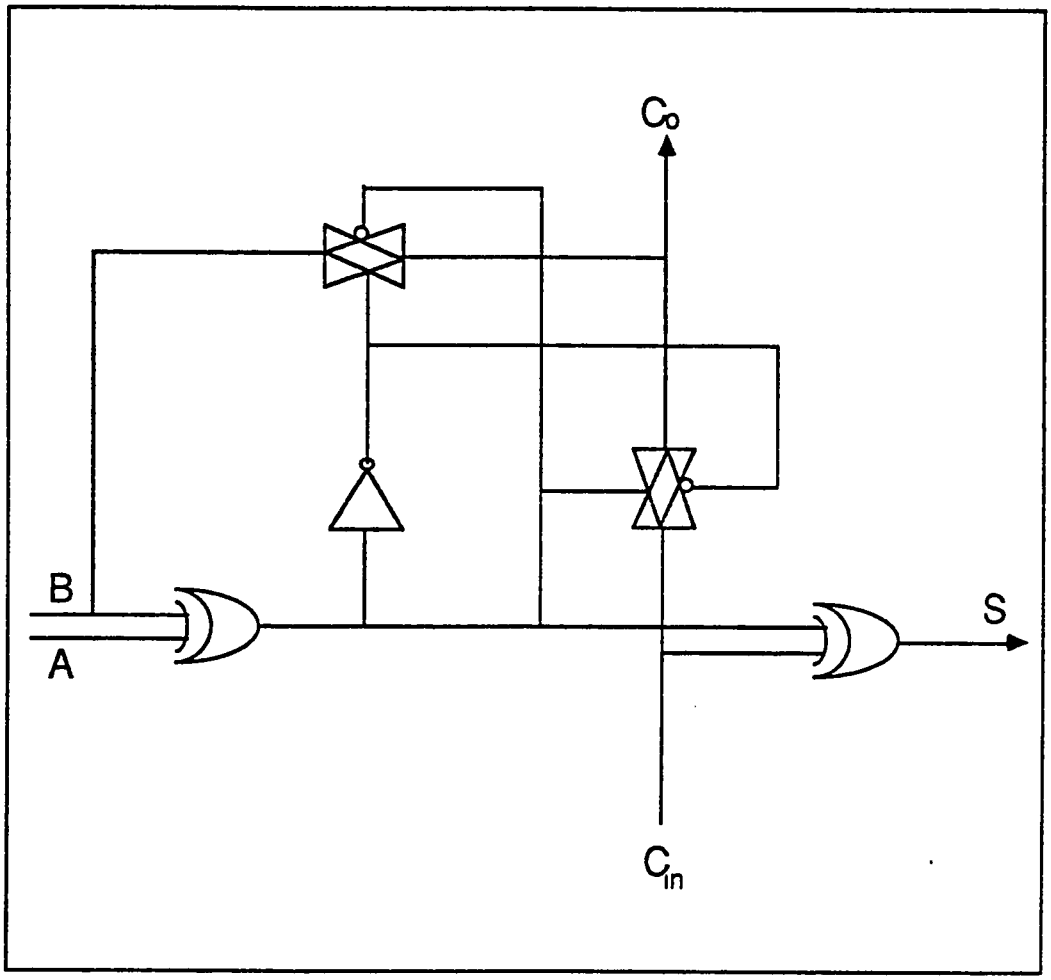


Figure 5.3: Logic diagram of a full adder

in this design because it avoides redundancy in the circuit, while a highly structured design, PLA for example, will result in unnecessary large size and delay.

A CMOS circuit diagram for the full adder is shown in Figure 5.4. A SPICE simulation for the transistor-level circuit of the 4-bit adder has been worked out to ensure functional correctness. The SPICE model of adder is provided in Appendix B (PROC 1, SUBCKT ADDER).

5.3.2 Magnitude Comparator Design

Logic diagram of a comparator is depicted in Figure 5.5. The comparator compares the two numbers $A = A_3\ A_2\ A_1\ A_0$ and $B = B_3\ B_2\ B_1\ B_0$. The algorithmic design of the comparator is a direct application of the procedure a person uses to compare the relative magnitude of two numbers. The equality relation of each pair of bits can be expressed with an equivalence function.

$$x_i = A_i B_i + \overline{A_i}\ \overline{B_i} \qquad i=0,1,2,3 \qquad \text{-------- (5.3)}$$

The sequential comparison of A and B can be expressed logically by the following Boolean functions:

$$(A<B) = A_3\overline{B_3} + x_3A_2\overline{B_2} + x_3x_2A_1\overline{B_1} + x_3x_2x_1A_0\overline{B_0} \qquad \text{-------- (5.4)}$$

$$(A=B) = x_3x_2x_1x_0 \qquad \text{-------- (5.5)}$$

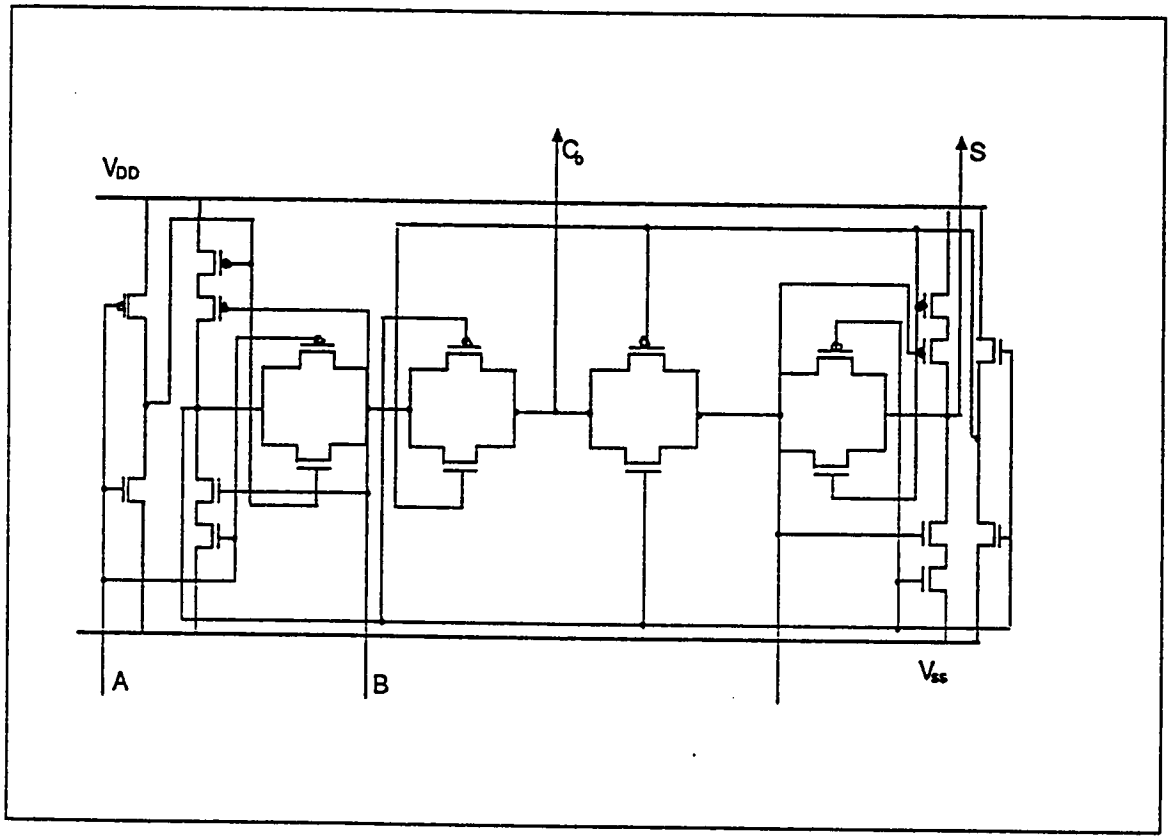


Figure 5.4 : Transistor diagram of a full adder

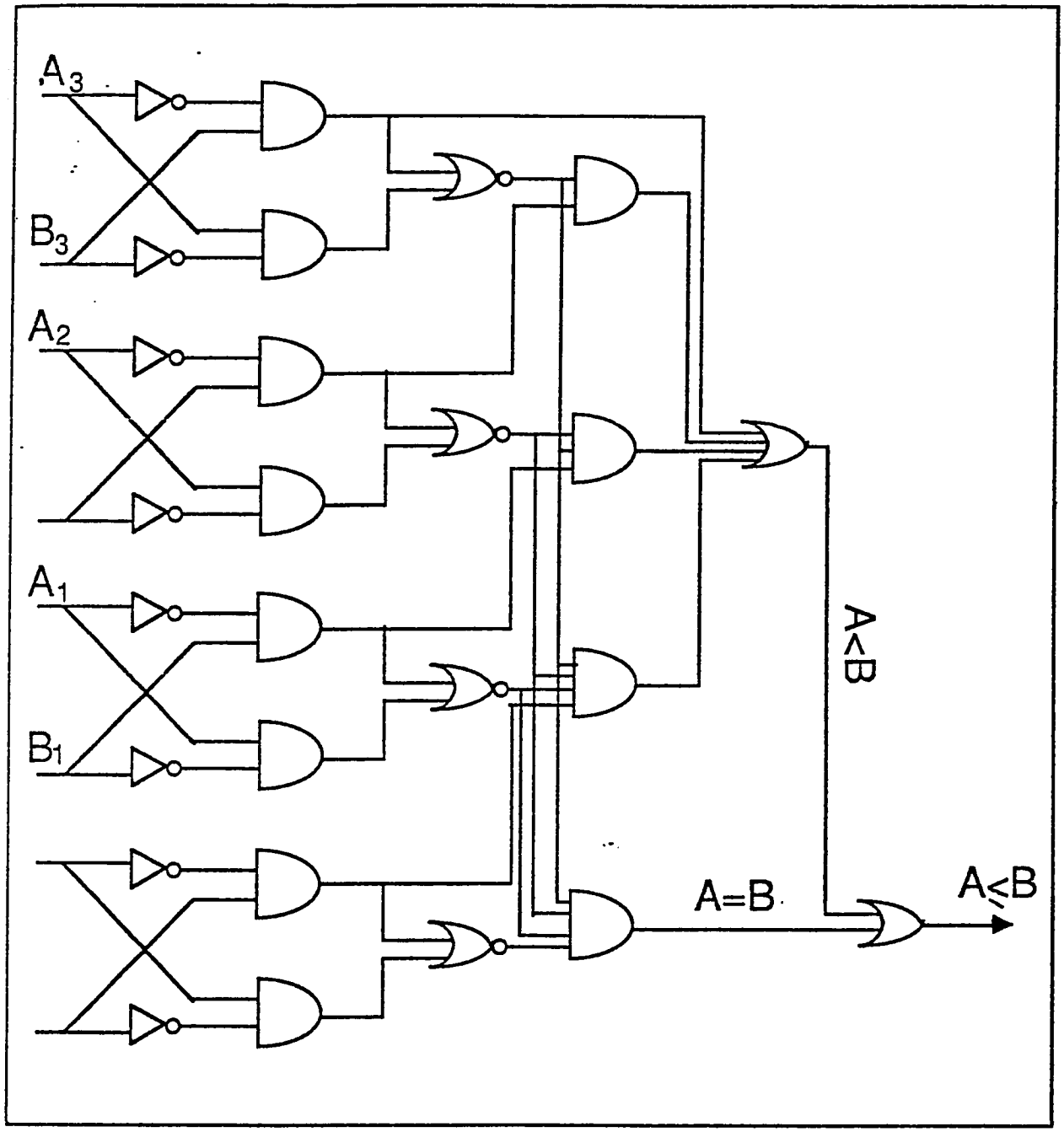


Figure 5.5: Logic Diagram of a 4-bit Comparator

It is to be noted that extension of the design to compare binary numbers with more than four bits is straight forward since regular algorithmic procedure is adopted for this design [24].

A CMOS circuit has been designed for four-bit comparator as shown in Figure 5.6. Note that in the transistor-level circuit a one-to-one mapping with the logic circuit of Figure 5.5 is not maintained. Small modifications has been made (See Figure 5.7) to sustain regular and repetitive structure of the design so as to ease the VLSI layout design. The transistor-level circuit of the comparator has been simulated using SPICE. SPICE model of the comparator is provided in Appendix B (PROC 1, SUBCKT CMPARTOR).

5.3.3 Hamming Distance Calculator

Hamming distance calculator HDC is a very simple combinational logic circuit which calculates the Hamming distance between the input and a constant sequence (α) which is directly derived from the trellis structure. As mentioned in the previous chapter the length of (α) is dependent on the rate. For the case of 1/2 rate code the length is two and the output of the HDC; denoted by $Z_3\ Z_2\ Z_1\ Z_0$, can be expressed logically by the following Boolean functions:

$$Z_3 = 0, \quad Z_2 = 0 \qquad \text{-------- (5.6)}$$

$$Z_1 = (I_1 \oplus \alpha_1).(I_0 \oplus \alpha_0) \qquad \text{-------- (5.7)}$$

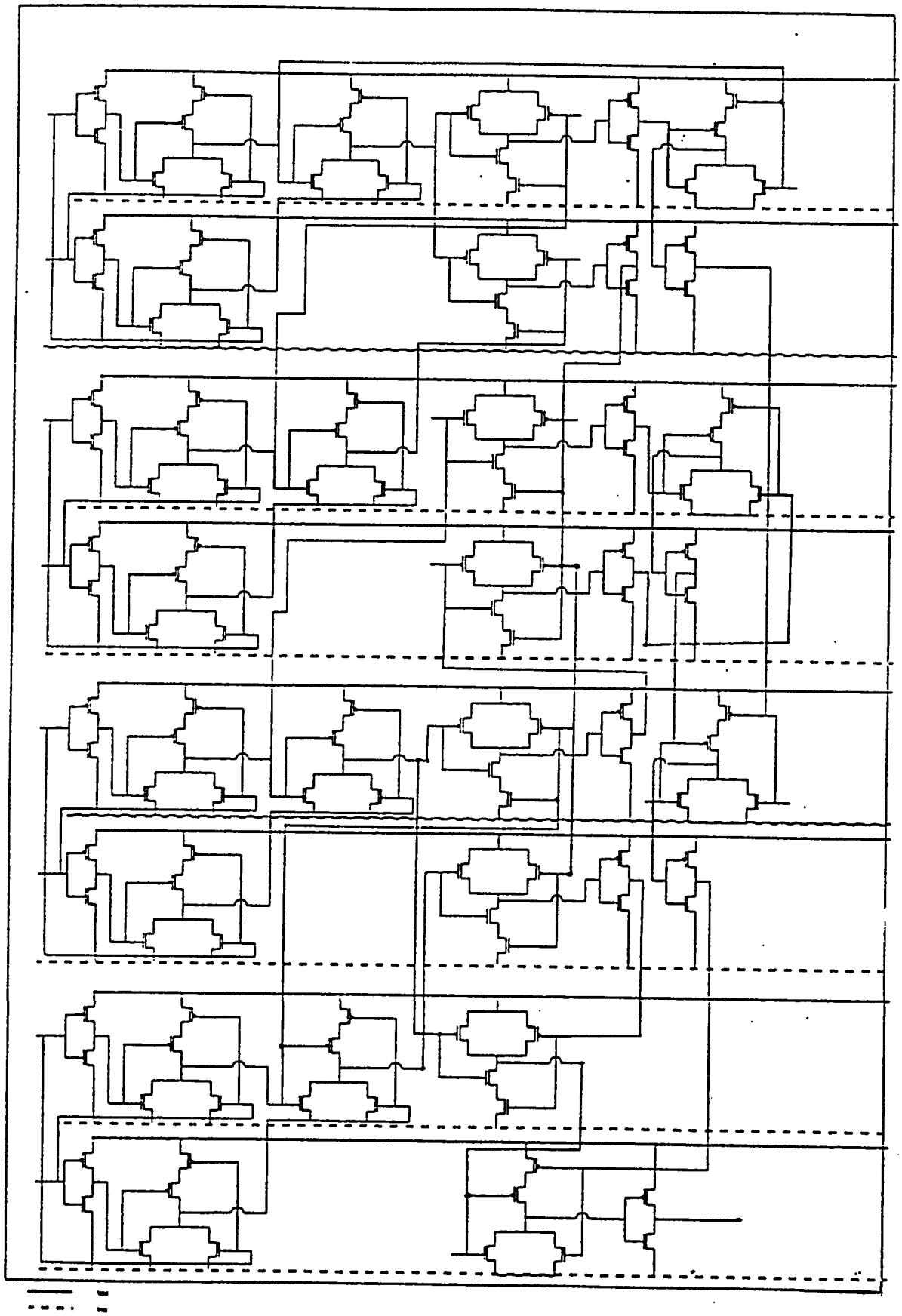


Figure 5.6 : CMOS circuit for a 4-bit magnitude comparator

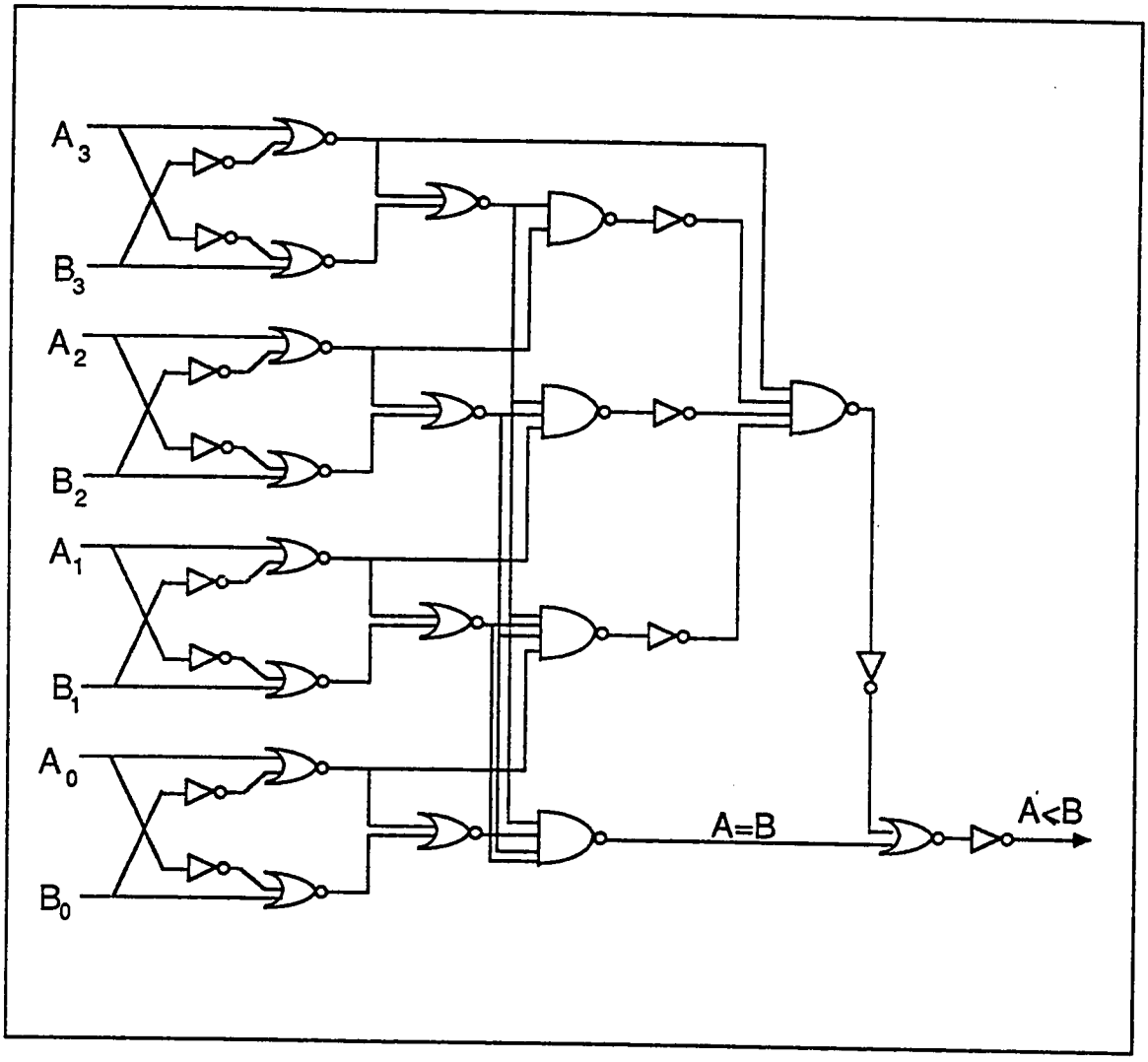


Figure 5.7 : Logic diagram of a 4-bit comparator

$$Z_0 = (I_0 @ \alpha_0) @ (I_1 @ \alpha_1) \qquad \text{-------- (5.8)}$$

where $I = I_1\ I_0$ indicates the sub-block input and $\alpha = \alpha_1 \alpha_0$ represent the constant sequence stored in the processor.

In order to avoid overflow in the path metric registers, subtracting a one from the content of each register in every clock cycle is adopted, provided that the contents of all registers are greater than zero. However, subtraction process adds hardware complexity, increases the silicon area and slows down the speed of the first block processors. To overcome this problem HDC subtracts a one from the Hamming distance obtained. This is done before adding the results of HDC to the path metric. It is to be noted that subtraction is accomplished using 2's complement. 2's complement is necessary for the case that the Hamming distance equals zero (In this case the output of HDC is the 2's complement of -1 which is 1111).

Instead of using one general kind of HDC to do the Hamming distance calculation and constant subtraction; which involves complicated circuitry, simpler circuits for HDC VLSI layout can easily be derived if each of the four possible values of α are evaluated independently. Boolean logical expressions for the output of HDC for different α's are as following:

i) For $\alpha = 00$

$Z_3 = \overline{I_1 \oplus I_0}$

$Z_2 = \overline{I_1 \oplus I_0}$

$Z_1 = \overline{(\overline{I_1} + \overline{I_0} + C) . (\overline{C} + I_1 + I_0)}.$

$Z_0 = C \oplus I_1 \oplus I_0$

ii) for $\alpha = 01$

$Z_3 = \overline{I_1 \oplus \overline{I_0}}$

$Z_2 = \overline{I_1 \oplus \overline{I_0}}$

$Z_1 = \overline{(\overline{I_1} + I_0 + C) . (\overline{C} + I_1 + \overline{I_0})}.$

$Z_0 = C \oplus I_1 \oplus I_0$

iii) For $\alpha = 10$

$Z_3 = \overline{\overline{I_1} \oplus I_0}$

$Z_2 = \overline{\overline{I_1} \oplus I_0}$

$$Z_1 = \overline{(I_1 + \overline{I_0} + C).\overline{(C + \overline{I_1} + I_0)}}.$$

$$Z_0 = C \oplus I_1 \oplus I_0$$

iv) For $\alpha = 11$

$$Z_3 = \overline{\overline{I_1} \oplus \overline{I_0}}$$

$$Z_2 = \overline{\overline{I_1} \oplus \overline{I_0}}$$

$$Z_1 = \overline{(I_1 + I_0 + C).\overline{(C + \overline{I_1} + \overline{I_0})}}.$$

$$Z_0 = C \oplus I_1 \oplus I_0$$

It is to be noted that in the above expressions 'C' equals the logic '1' if none of the registers in all processors of the first block has an all zero content which indicates that subtracting one from all registers should take place. The logic diagrams and the corresponding CMOS circuit diagrams of HDC for different α's are shown in Figures 5.8, 5.9, 5.10 and 5.11. CMOS circuits for basic elements of HDCs are depicted in Figure 5.12. All CMOS circuit diagrams have been simulated using SPICE.

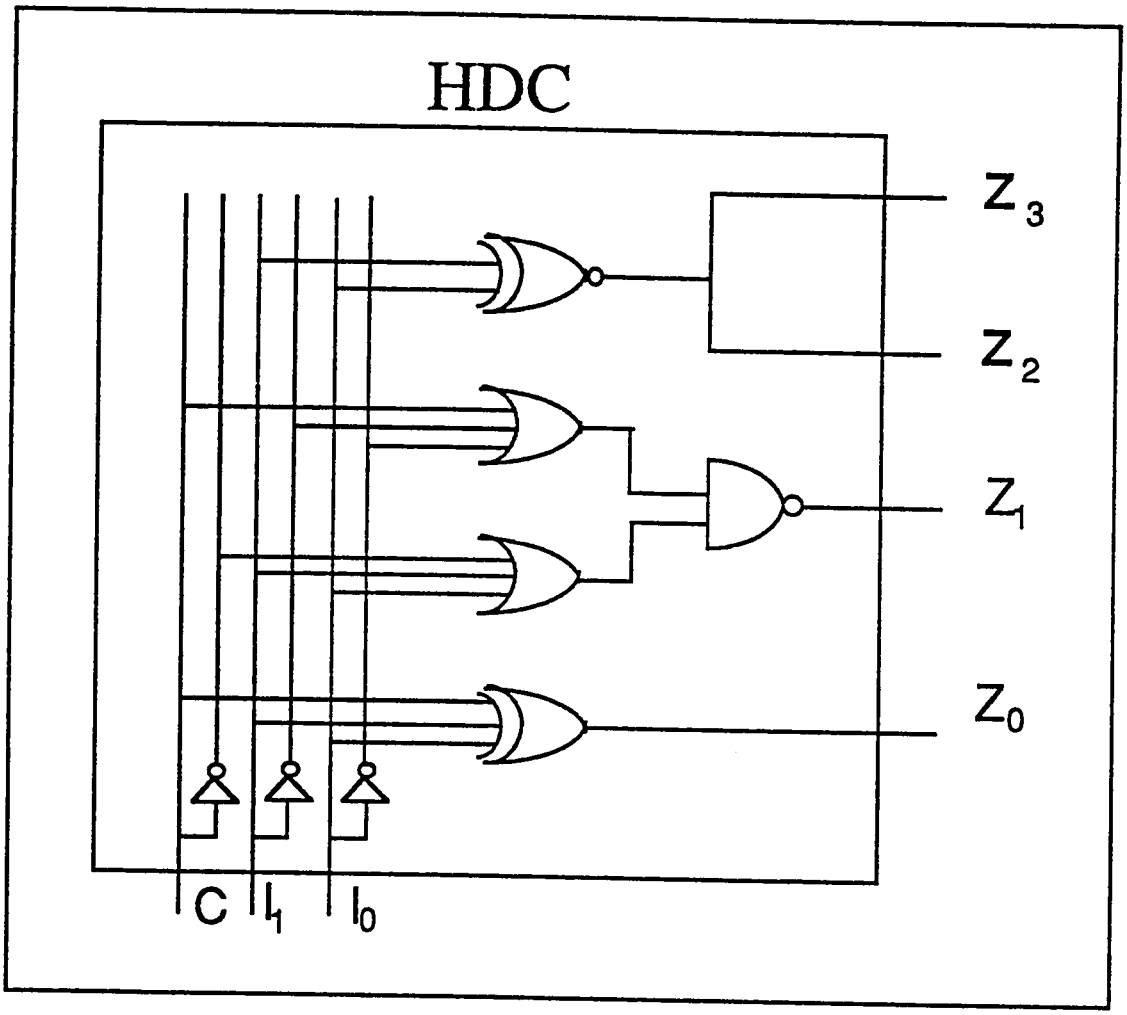


Figure 5.8 : Logic diagram of HDC for $\alpha = 00$

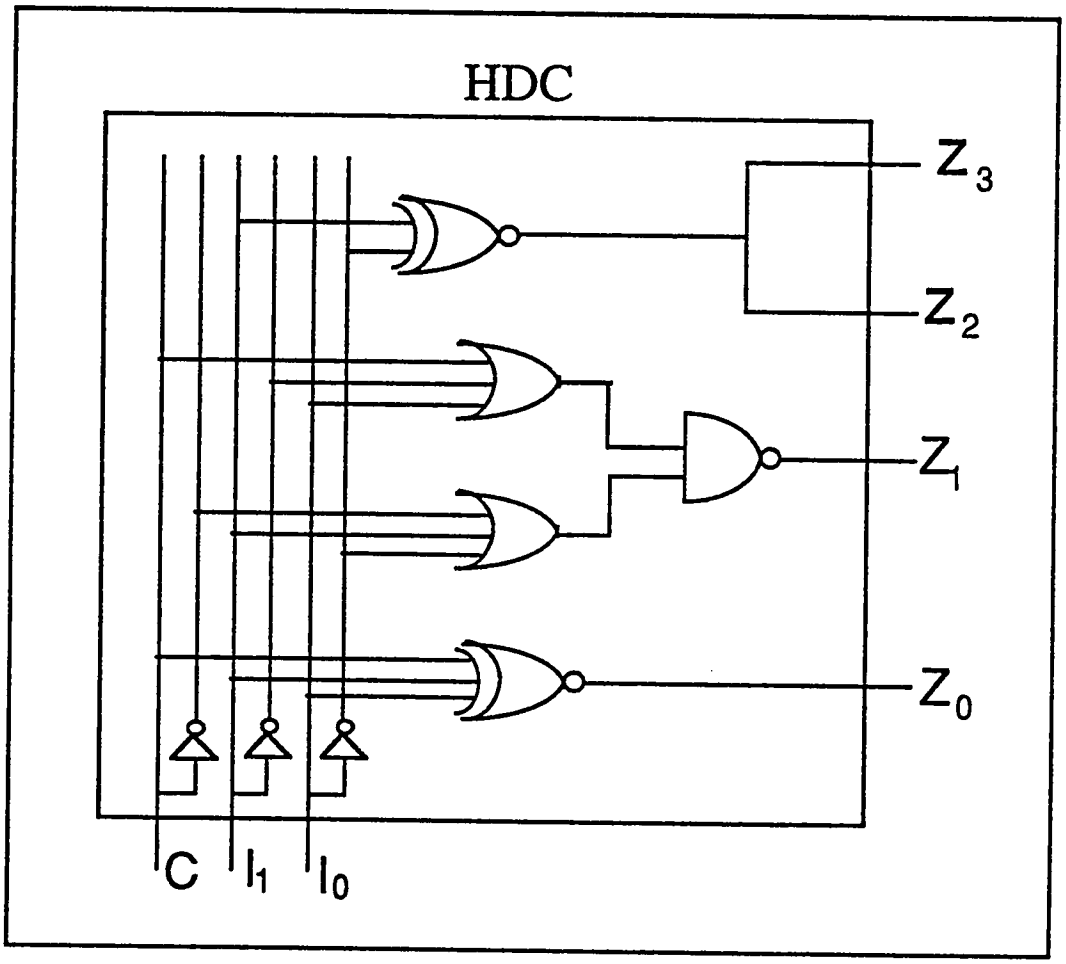


Figure 5.9 : Logic diagram of an HDC for $\alpha = 01$

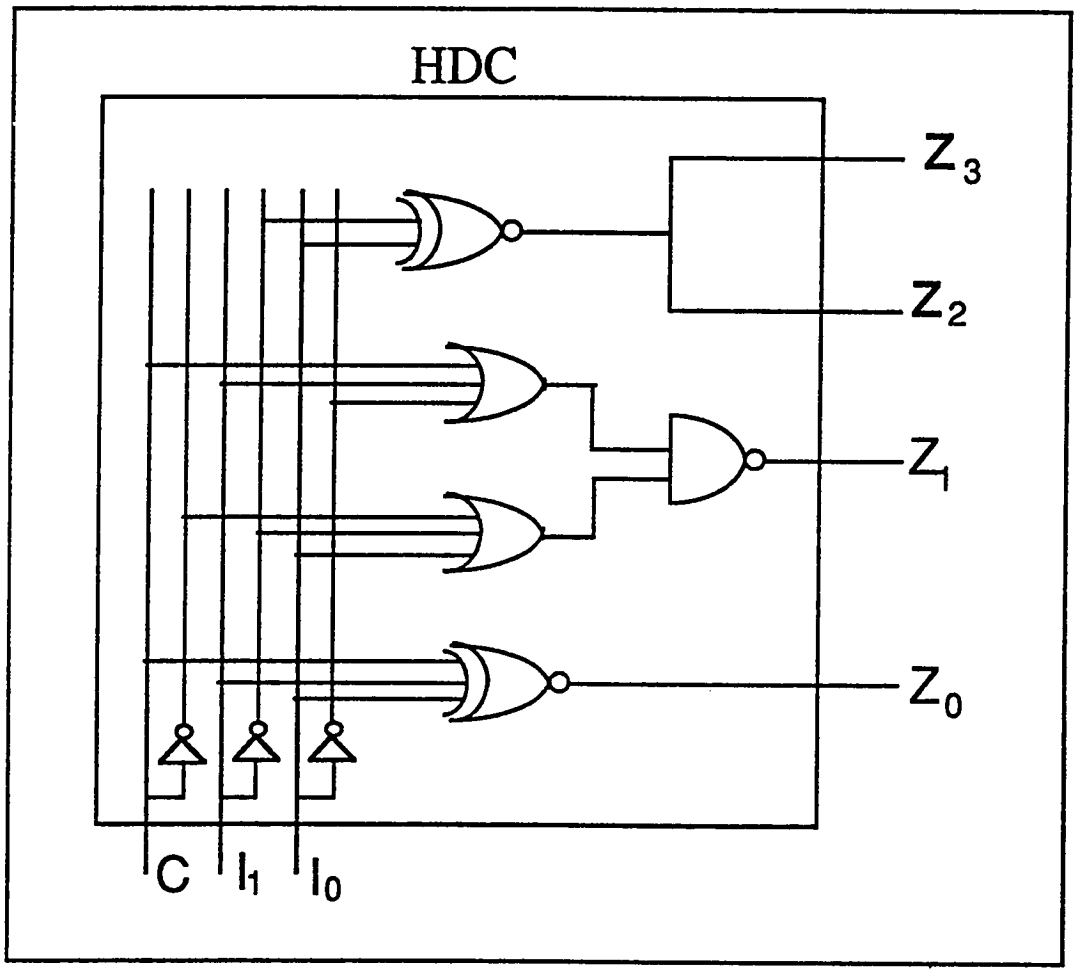


Figure 5.10: Logic diagram of an HDC for $\alpha = 10$

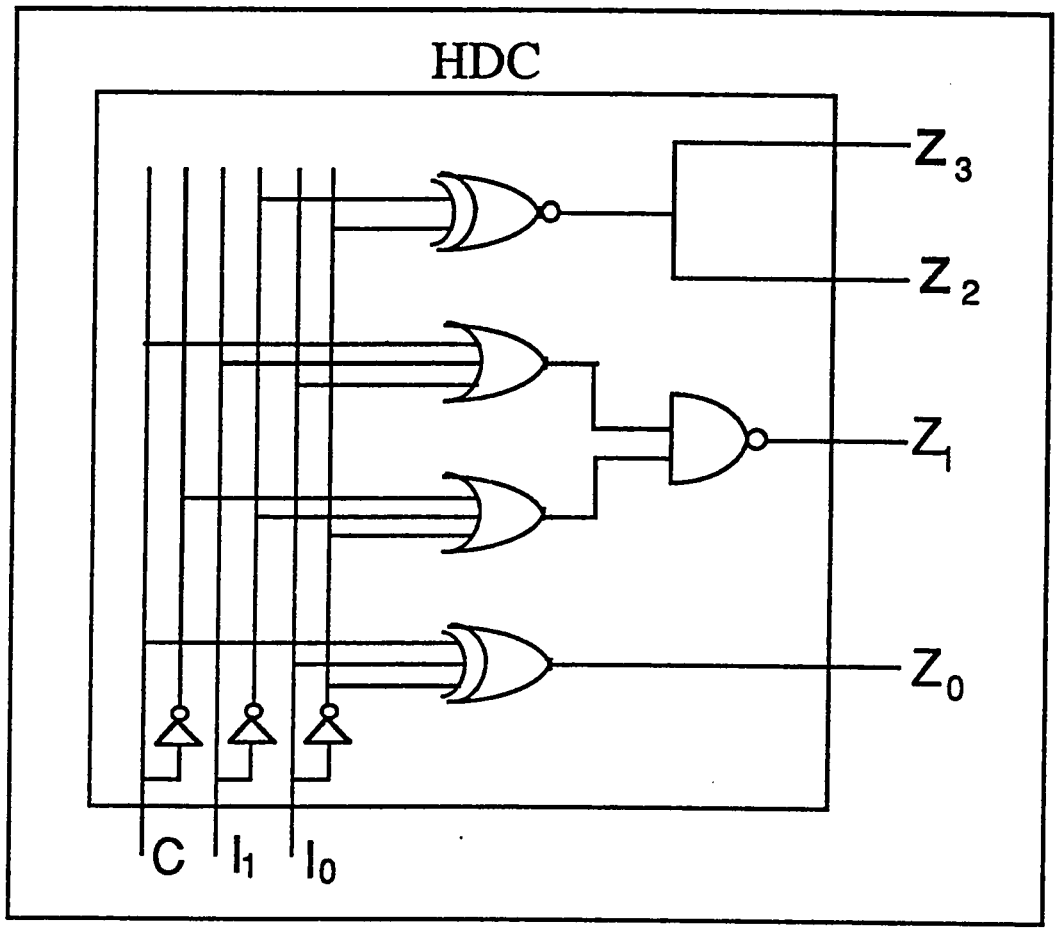


Figure 5.11 : Logic diadram of an HDC for $\alpha = 11$

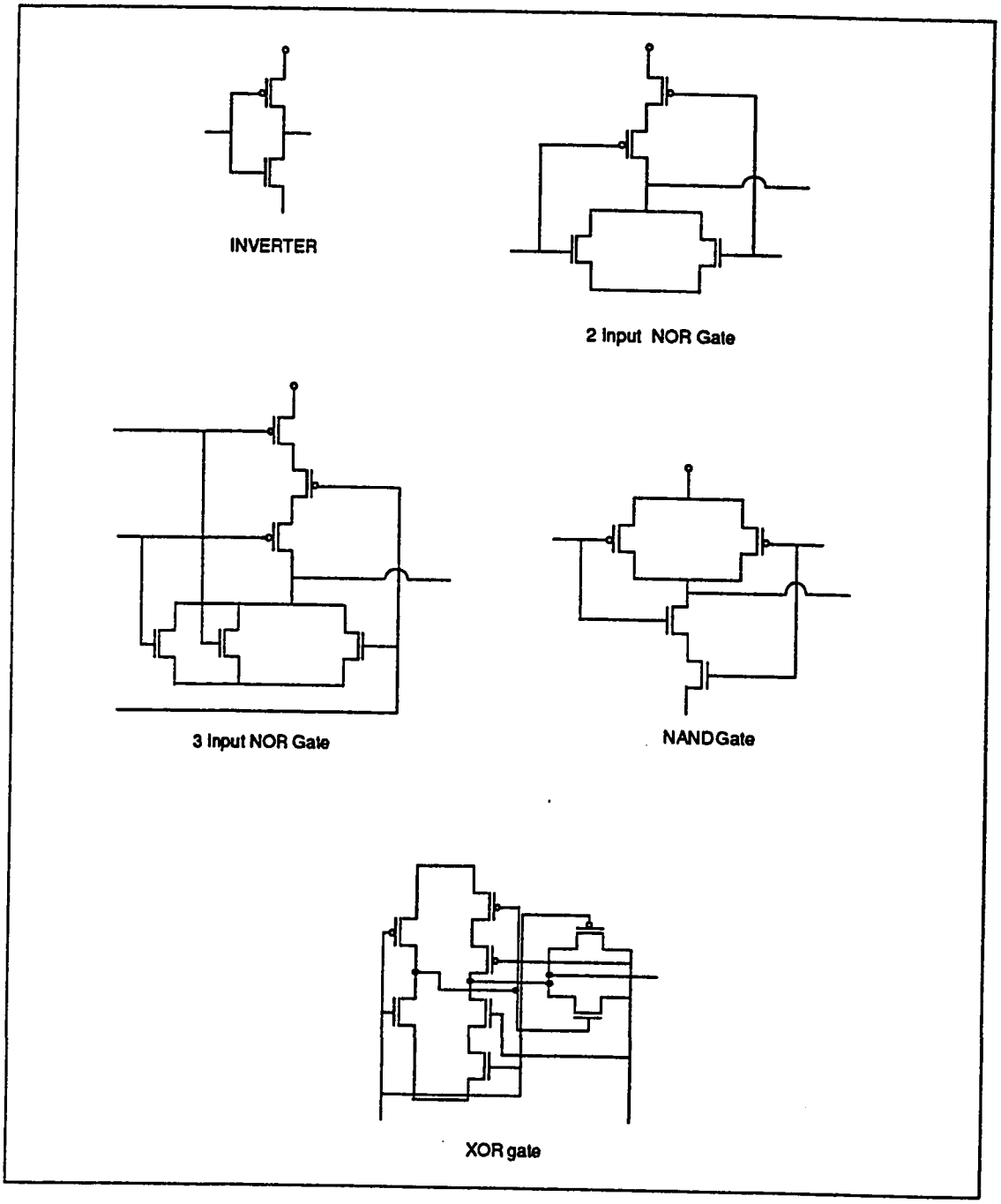


Figure 5.12 : Basic Gates Used in HDCs

5.3.4 Operation of the First Block

As shown in Figure 5.13 during the second phase (φ_2) of the clock period the Hamming distance is computed in both HDCs of each processor. At the same time the comparison between the previous path metrics is performed by the comparator. The output of the comparator S is connected to the select line of the multiplexer. Then according to the value of S (0 or 1), the path with the minimum metric is selected by the multiplexer and is transferred; during φ_1 of the clock period, to two adders of processors whose address is the "shuffle" and "shuffle - exchange"; respectively, of the binary address representation of the source processor. At the same time, the flag corresponding to the survived path is set to "1" in the source processor, while the other flag is cleared. This process is repeated every clock cycle where new input is received. At φ_1 of every clock cycle the contents of all flags of processors in the first block are transferred to the flags of the second block.

The interconnection between processors of the first block is shown in Figure 5.14.

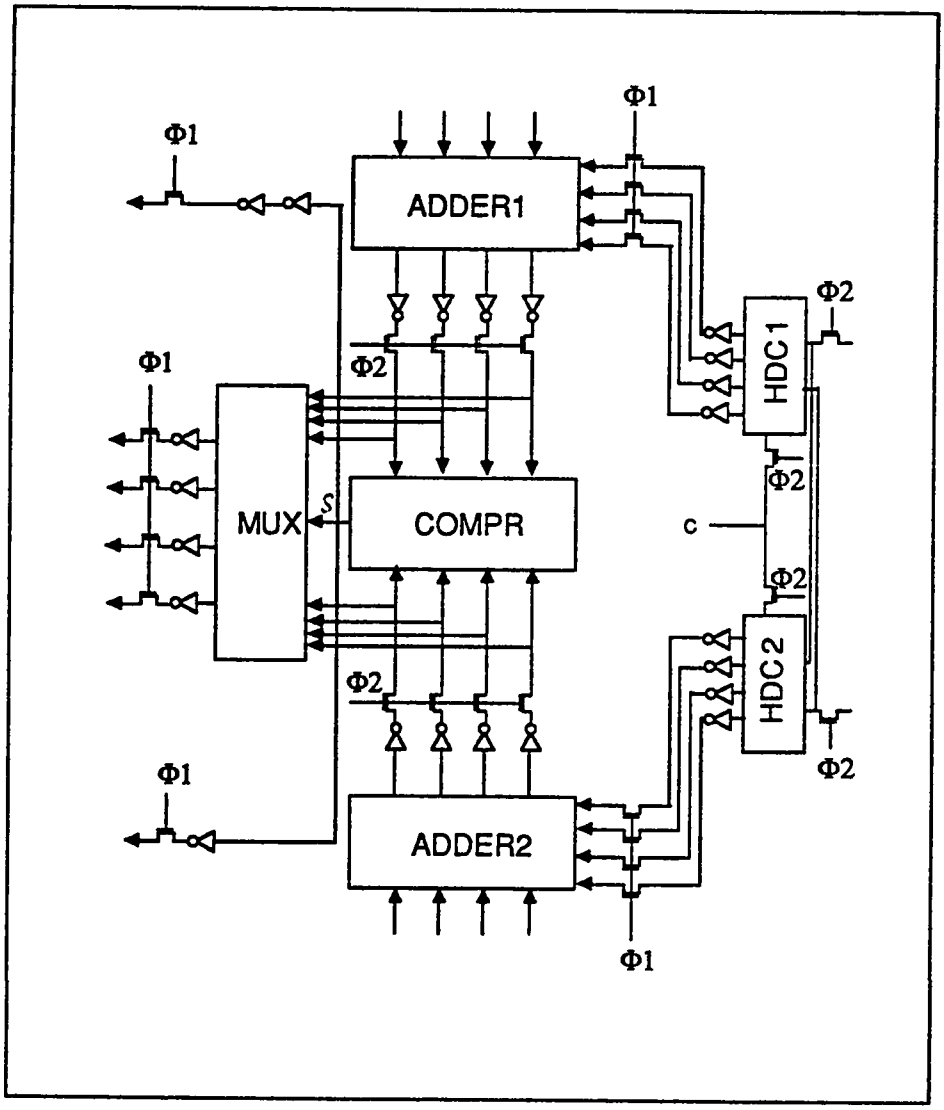


Figure 5.13 : Design of a processor of the first block

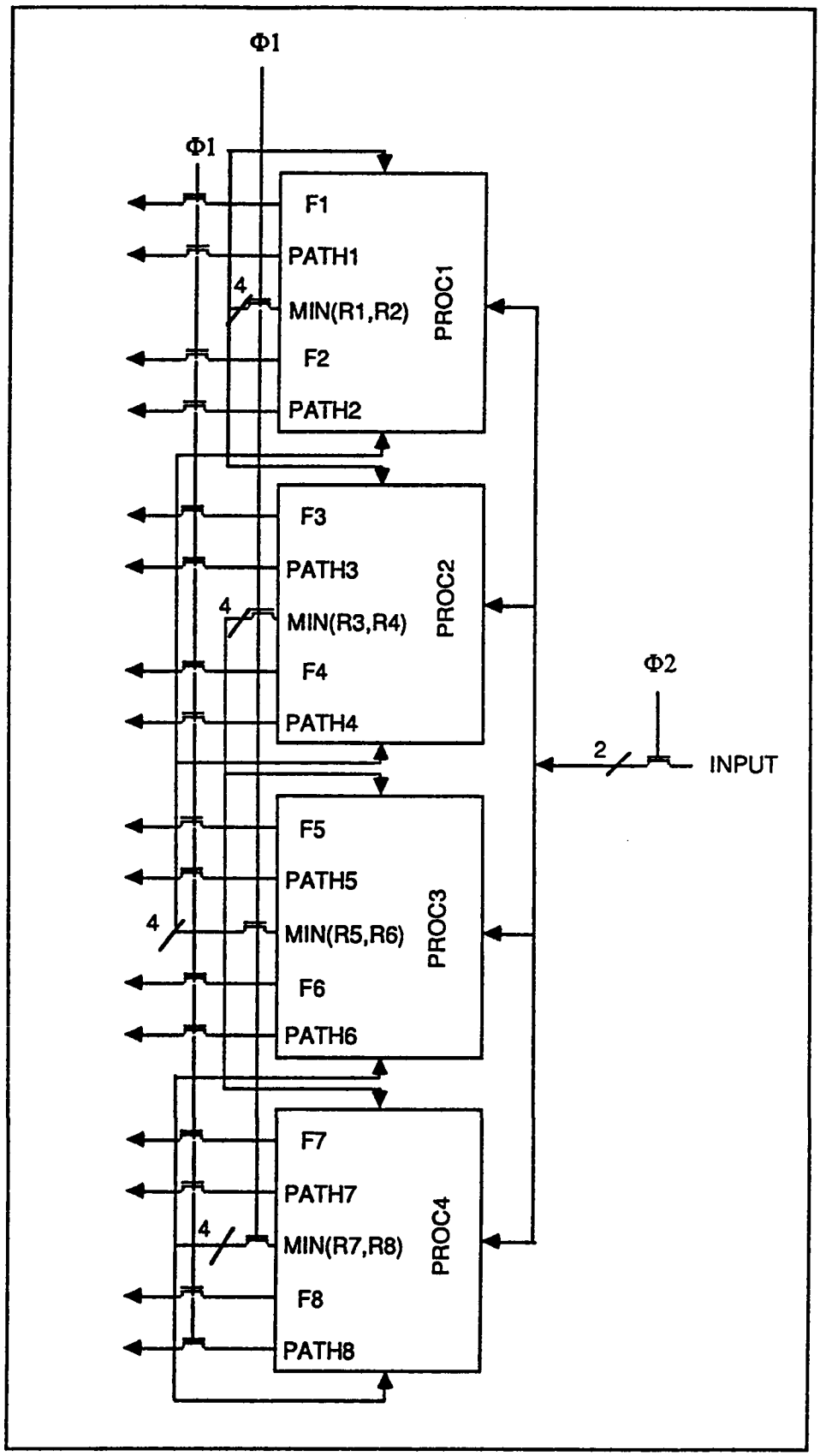


Figure 5.14 : Interconnection between processors of the first block

5.4 DESIGN OF THE SECOND BLOCK

Design of second block processors is very simple. A processor ij receives the contents of flags (F_1, F_2) of processor i(j-1), also it receives the contents of another two flags from two processors whose binary address representation is the "shuffle" and "shuffle-exchange" of its binary address representation. The second block processors take care of the path history; in addition, they keep track with the survived path updating. This is accomplished by checking the status of each node of the trellis. For any node, if there is no survived branch connecting it with other nodes; which indicates that the path history is no longer a survived path, then the flags of the processor corresponding to that node should be both cleared and should also clear the whole path history. The design of a processor of the second block is shown in Figure 5.15. The interconnection between the first and second block processors is depicted in Figure 5.16.

It is to be noted that the number of columns in the second block represents the length of the path history to be stored.

5.5 CMOS VLSI IMPLEMENTATIONS OF CELLS

The architecture of a Viterbi decoder presented in this work is well-suited for VLSI implementation since the decoding process is partitional among a large number of identical units.

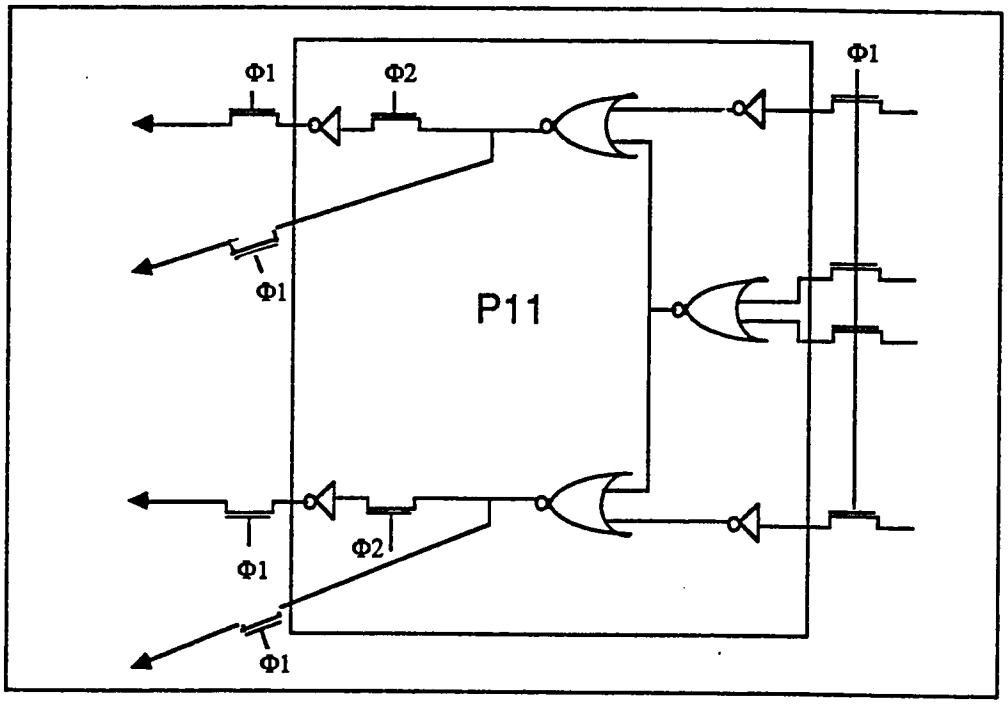


Figure 5.15 : Design of a second block processor

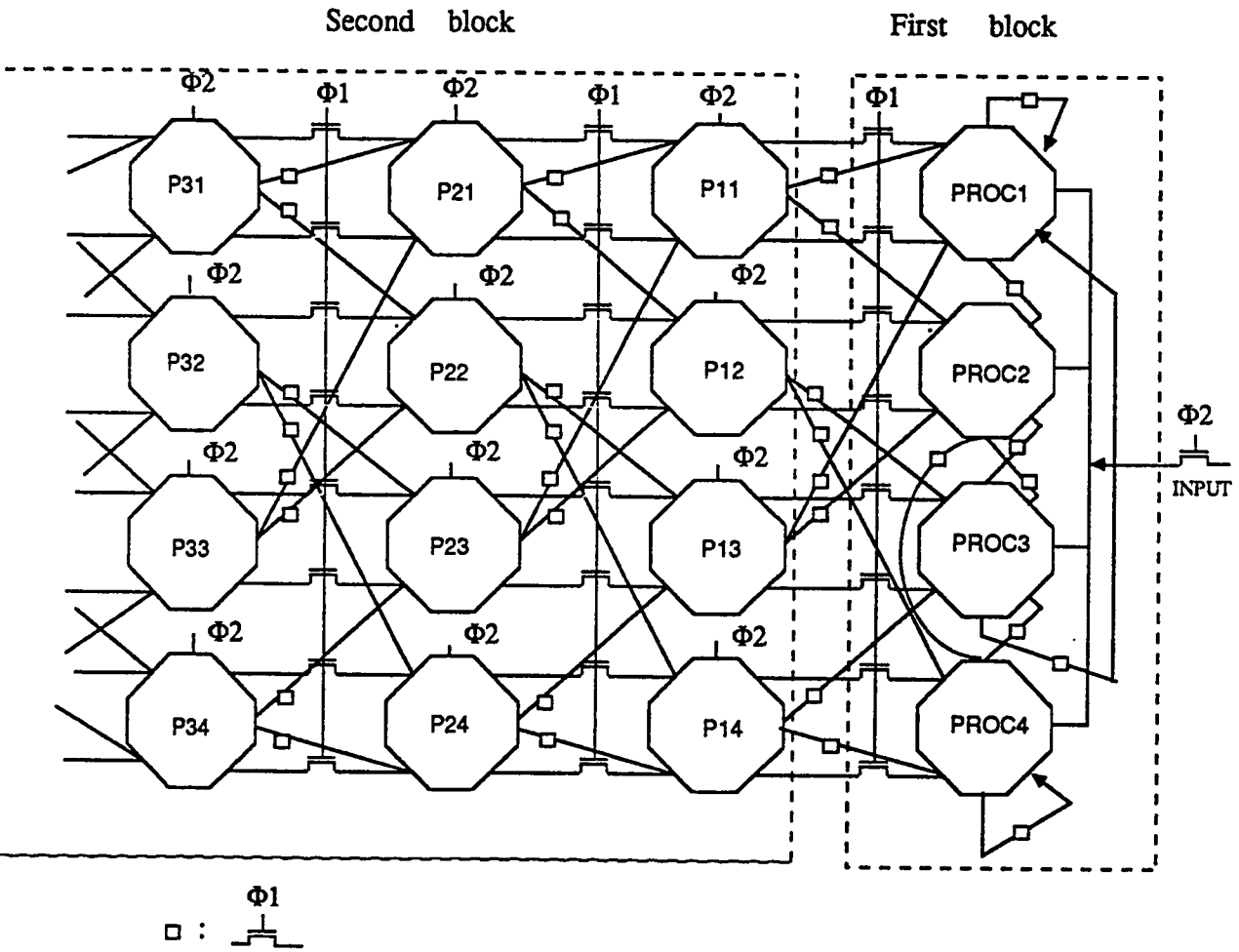


Figure 5.16 : Interconnction between processors of the first and the second block

In this section the CMOS VLSI implementations for different cells of the decoder are presented. The data format and symbolic language chosen for CMOS VLSI design is LUCIE [16]. Design rules of IMAG/TIM3 [16] are used in the design of layouts. Figure 5.17 depicts the VLSI layout of the full adder shown in Figure 5.3.

LUCIE files for the adder and the 4-bit magnitude comparator are shown in Appendix C (ADDER, COMPARATOR).

5.6 PLACEMENT OF CELLS ON THE LAYOUT FLOOR

Efficient placement of cells on the layout floor is very important to achieve optimal size of the overall circuit. The silicon area consumed by any circuit has to be minimized to permit large device capacity in a VLSI chip. Circuit layout of different cells of the two blocks are used to determine the area of the circuit. Dimensions of the cells are tabulated in Table 5.1. It is to be noted that the height and width of different cells are directly measured from the layout in terms of λ where λ is a unit spacing factor which varies according to the advances in mask alignment techniques. Using design rules of IMAG/TIM3 λ is considered to be 2 μm.

Floor plan for the 4-bit magnitude comparator is depicted in Figure

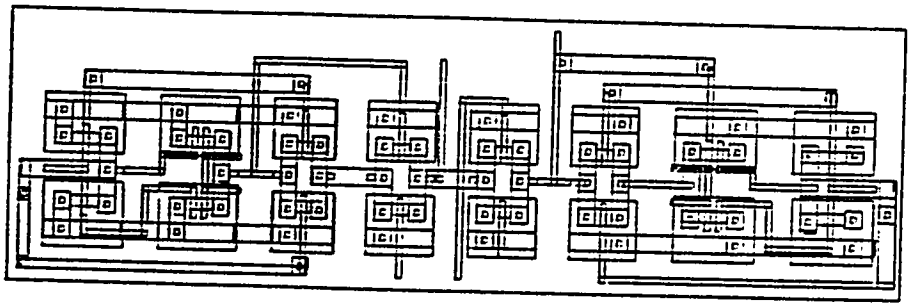


Figure 5.17a : VLSI layout of 1-bit full adder

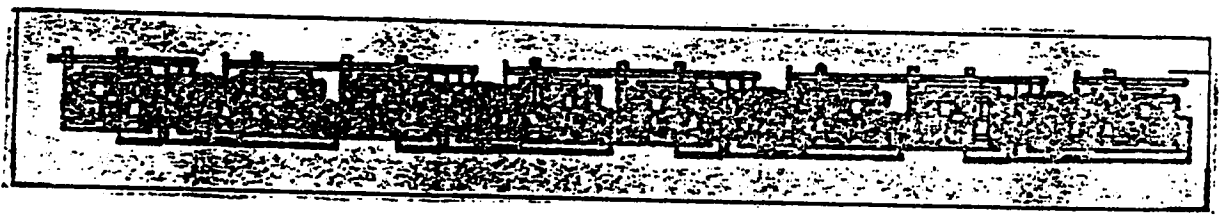


Figure 5.17b : VLSI layout of 4-bit adder

Element	Height	Width
N-pass tran.	29	46
INVERTER	54	29
NAND	58	36
NOR	56	42
1-bit Adder	130	340
4-bit Adder	130	1350
Comparator	550	350
Processor 1	750	1350
Processor 2	280	230

Table 5.1: *Dimentions of cells mesured in* λ

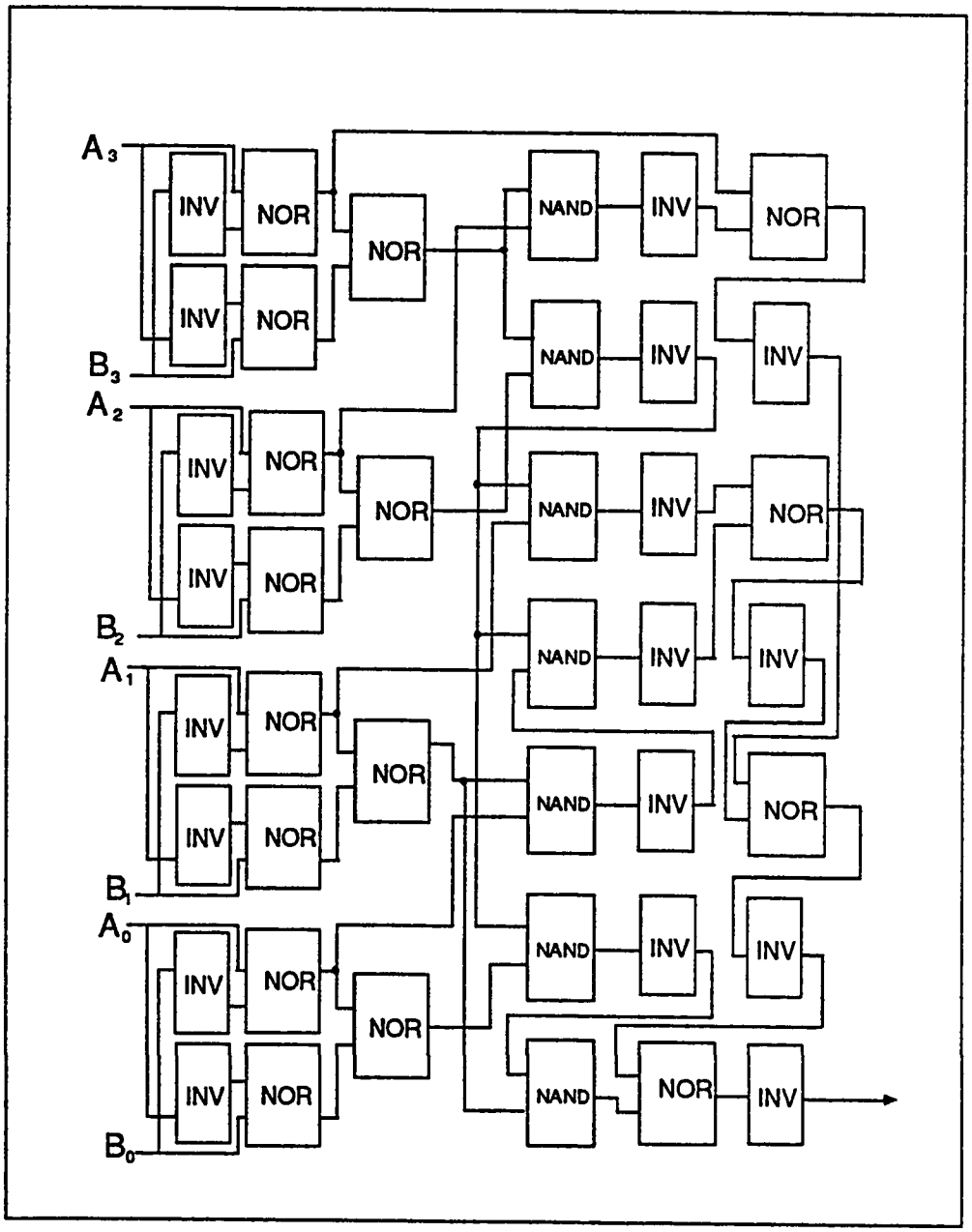


Figure 5.18: Floor plan for a 4-bit comparator

5.18 where relative sizes of logic gates are considered. The area occupied by the comparator is 350 $\lambda \times 550\lambda$. The area occupied by first processor is 1350 $\lambda \times 750\lambda$ using IMAG/TIM3 rules.

CHAPTER VI

ANALYSIS AND PERFORMANCE EVALUATION

6.1 INTRODUCTION

The large number of applications of the Viterbi algorithm in digital data communication systems has encouraged many researchers to look for practical implementations for it. In this chapter a survey on existing architectures of Viterbi decoders and a comparison with two of these with the systolic decoder designed in this research are provided. A measure for evaluating the performance of any VLSI design for Viterbi decoders is also discussed. The next section (6.2) provides a methodology for extending the capability of our designed systolic Viterbi decoder for any k/n rate convolutional code.

6.2 SYSTOLIC VITERBI DECODERS FOR k/n RATE CONVOLUTIONAL CODES

So far we have dealt only with 1/2 rate convolutional codes which are the most commonly used convolutional codes in practical applications. Generalization to code rates other than 1/2 are straight

forward. The rate R = 1/n case is the simplest since the same encoder structure is retained, but the two modulo-2 adders shown in Figure 2.2 are now replaced by n modulo-2 adders. Increasing the number of modulo-2 adders causes an increase in channel symbols for each branch of the trellis structure from 2 symbols into n symbols. The architectures of 1/n rate systolic decoders are identical to those of 1/2 rate decoders except that an appropriate increase of number of bits of HDC's, adders, comparators and registers are needed.

In order to avoid overflow, subtraction of a constant in every clock period is needed. The value of the constant to be subtracted depends on the length of registers, adders and comparators. The expected severity of error is also a determining factor in the selection of the constant.

The situation is more cumbersome for R = k/n codes with k > 1. A typical case is depicted in Figure 6.1 for R = 2/3, m = 4 code. In this case two symbols are shifted into the encoder for each branch. A trellis structure for this code is shown in Figure 6.2. It is to be noted that there are four branches entering each node instead of two. If a k/n convolutional decoder is to be implemented using the systolic algorithm, then the number of HDC's required is 2^k, the number of additions required is 2^k and the number of comparisons is $2^k - 1$. It is clear that this is a fairly serious implementation difficulty, particularly at high data rates.

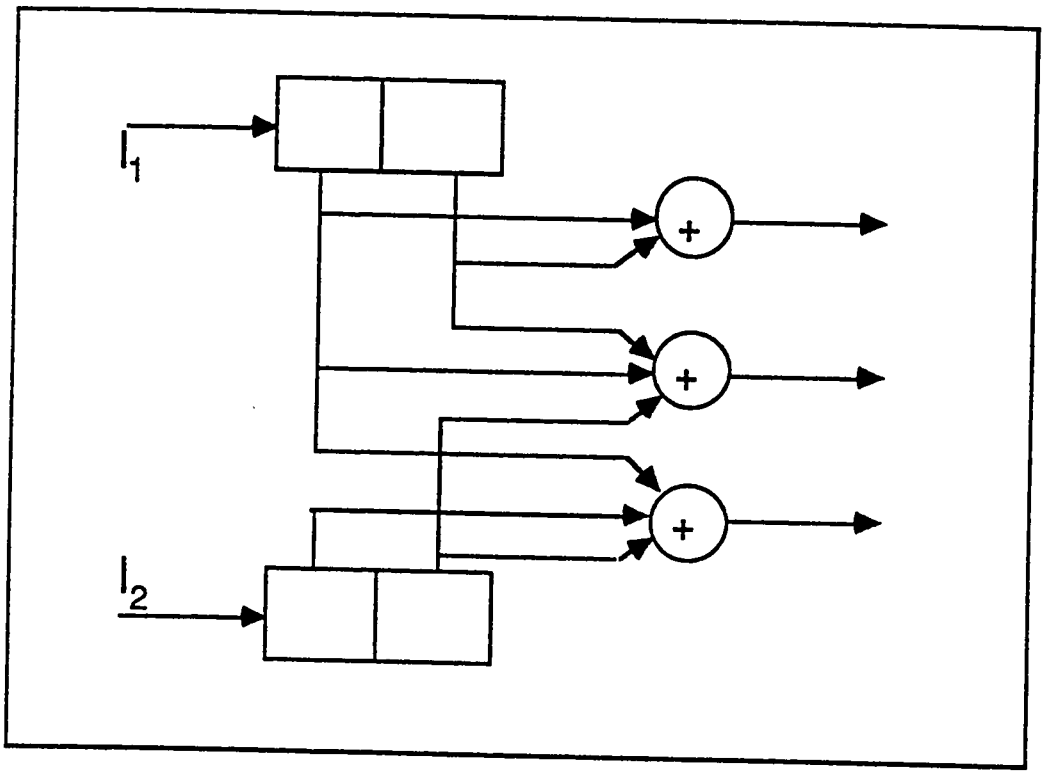


Figure 6.1 : Encoder for a Rate 2/3 , m=4 Code

The key to simplifying the implementation is to use the so called "punctured convolutional codes" [3]. The new code takes a R = 1/n code and "punctures" or deletes certain channel symbols thereby producing a R = k/n code. In order to illustrate this technique let us refer to Figure 6.2, where a rate 2/3 code is used. Now consider K = 1/2, m=3 code. Suppose every fourth encoder output bit is deleted, this code will produce three channel bits for every two data bits, i.e., it will be a R = 2/3 code. This code has the trellis shown in Figure 6.3 where X indicates the deleted symbols. The practical value of the punctured code approach is obvious. In our case, using the direct method requires three 2-ary comparison at each node for each three channel symbols while the punctured code approach uses two levels of 2-ary comparison. Therefore, one can implement an R = 2/3 decoder as an R =1/2 decoder with additional control to stuff erasures in the locations of the deleted bits. After the erasures are stuffed, decoding proceeds just as if the code were an R = 1/2 code. In this fashion we replace the complex 2^{n-1} -ary comparison at each state by binary comparisons.

6.3 SURVEY ON EXISTING VITERBI DECODERS

Many architectures for Viterbi decoders have been proposed in the literature. Said and Dimond [33] proposed a software implementation of the Viterbi algorithm on the MC68000 micoroprocessor. Conan and Oliver

[6] suggested a general class of machines based on microcomputers (or minicomputers). The machine suggested has been programmed on a DEC - PDP/11-20 minicomputer using MACRO-11 assembly language. Hard-wired implementations for Viterbi decoding are also available in the market such as Linkabit LV7026 decoder [11] which is capable of operating at up to 2 Mbits/s. This decoder is a 1/2 rate decoder with constraint length m = 7. A total of 356 TTL integrated circuits have been required for performing all functions.

A recent implementation of a Viterbi decoder for a digital communications system with a time-dispersive channel has been proposed in [14]. It is capable of operating at a data rate of 2400 bits/s. It is designed for a 1/2 rate convolutional codes with constraint length m = 10. The decoder consists of 5 chips. The first chip takes care of transition metrics calculation, the second chip calculates the state metrics, the third is responsible for path history, and the fourth and fifth chips are storage systems (memories) to store the old state metrics and new state metrics, respectively. Figure 6.4 depicts the block diagram of the decoder.

If we compare our systolic decoder with Linkabit LV7026 decoder the following conclusions can be drawn:

1) *Area considerations*: As mentioned earlier LV7026 decoders are 1/2 rate convolutional decoders with m=7. If we attempt to design a CMOS systolic decoder with the same specifications as

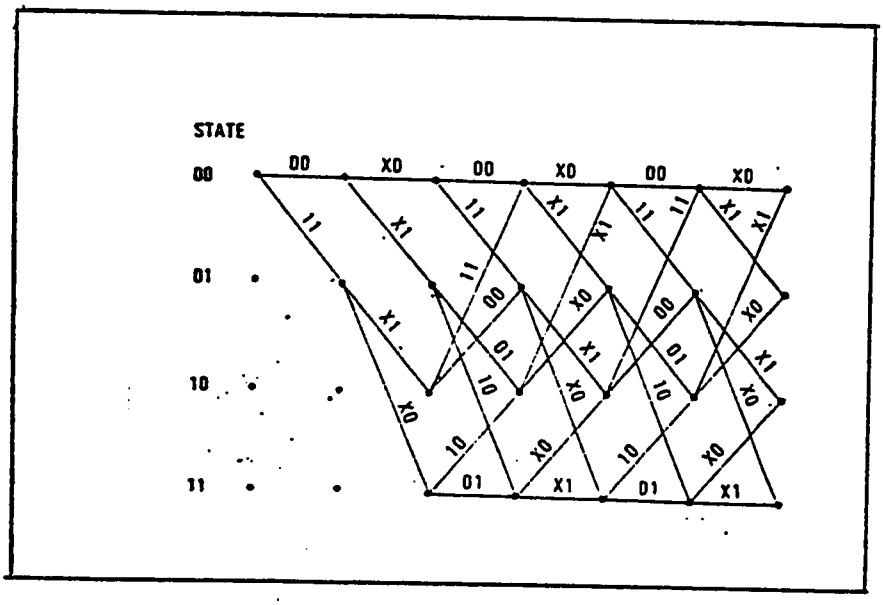


Figure 6.3 : Trellis structure of a rate 2/3 m=4 code produced by periodically deleting symbols from a rate 1/2 m=3 code

that of LV7026, then area occupied by the decoder is as following:

Area occupied by the first block processor = $N * A * R$

$= 2^{m-k} * A * R$

$= 2^{6} * [\ 1350\ \lambda * 750\ \lambda\ ] * 1.3$

$= 3.37\ cm^2$

Area occupied by second block processor = $L * N * A * R$

$= (4 * 7) * 2^{6} * [\ 280\ \lambda * 230\ \lambda\ ] * 1.3$

$= 6.00\ cm^2$

Where N indicates number of processors in a column, A represents the area occupied by each processor, R is a factor to compensate for increase of area due to routing, and L is the total number of processors in every row of the second block. Usually L is taken to be four times the constraint length of the code (i.e., 4*m). It is clear from above that the systolic decoder needs only few chips for fabrications, while in the case of LV7026 decoder 356 integrated circuits were needed. We can conclude that the systolic decoder occupies much less area than that occupied by the Linkabit LV7026.

2) *Power consumption*: It is well known that integrated circuits fabricated using TTL technology suffer from high power dissipation; while CMOS circuits enjoy minimum dissipation.

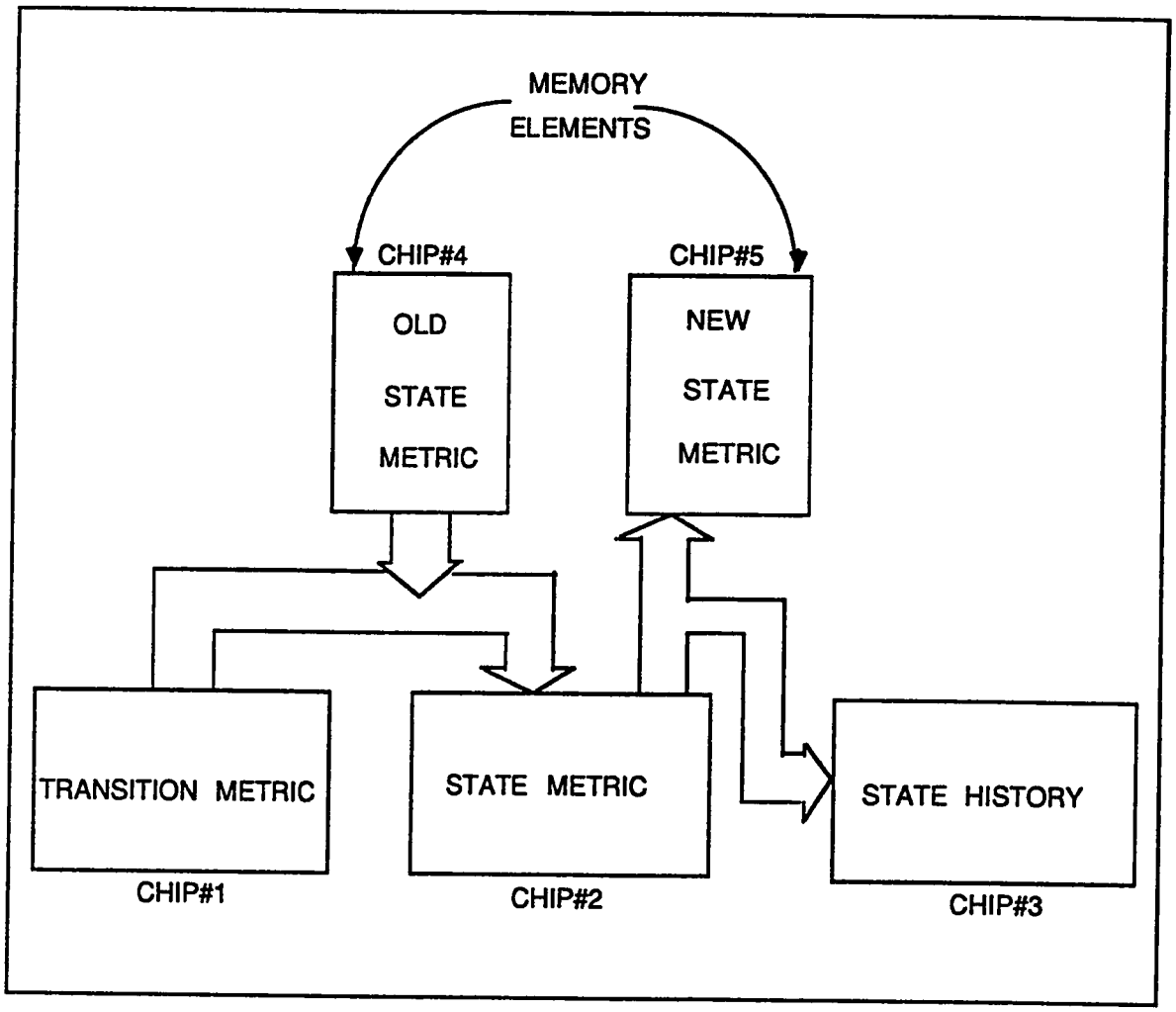


Figure 6.4 : Block diagram of the normal Viterbi decoder

Hence, the systolic decoder is better than LV7026 decoder.

3) *Speed*: Although TTL circuits are usually faster than CMOS circuits, the parallel processing approach followed in the design of systolic decoder, and maximum utilization of components make the systolic decoder faster than LV7026. A transient analysis of a 1/2 rate systolic decoder, using SPICE simulator, showed that a 1/2 rate systolic decoder can operate up to 12 Mbits/s approximately. Transient analysis for different cells of the systolic decoder are shown in Appendix B. The results of the transient analysis are tabulated in Table 6.1.

Hence, we can conclude that the designed systolic decoder is faster, consumes less power and is lighter, in terms of weight, than the LV7026 decoder.

If a comparison is made between the systolic decoder and the "5-chip" decoder mentioned above, which we will call the "normal decoder" from now on, then we can draw the following conclusions:

1) Both decoders use the same fabrication technology, namely CMOS.

2) The systolic decoder suits systems with very high speeds, such as space communication systems, while the normal decoder is designed for low rate communication systems (2400 bits/s). therefore comparing the two architectures is somehow missing the

Cell	Max. delay (nano-sec.)
4-bit full adder	60
4-bit comparator	8
Multiplexer	6
HDC	4
Total delay	80

Speed	12.5 Mbits/sec.

Table 6.1: *Results of transient analysis for different cells*

common ground.

3) Area occupied by the systolic decoder is higher than area occupied by the normal decoder provided that same specifications of constraint length and code rate that are used for the design of the normal decoder are used in the systolic decoder design. In fact, if calculations of area are performed in the same way as shown above. Then the total area needed for systolic decoder is about 98 cm^2, while area occupied by the normal decoder is less than 10 cm^2.

It is clear from above analysis that there is an area-speed trade off between the two designs. The one which sacrifices speed maintains a compact area, while the one which sacrifices area can operate at higher speeds.

If a closer look is made on the architectures of the two decoders, it can be observed that in the systolic decoder old state metrics and new state metrics do not need to be stored in external storages, as in the case of the normal decoder, but instead the metric for each state is stored in the corresponding processor of the first block. This feature enhances the performance of the systolic decoder. Another advantage of the systolic decoder is that it can easily be improved to decode for any k/n rate code, while in the case of the normal decoder this feature is much harder to reach.

Comparisons made on the systolic decoder on one side, and the normal and LV7026 decoders on the other side show that the systolic decoder pocesses many features that are not available in the other two decoders. Table 6.2 shows area occupied and operating speed for the three decoders. The above discussion shows that the systolic decoder is a fairly good candidate for use in space communication systems where low power consumption and light weight are needed.

6.4 AREA-TIME COMPLEXITY MEASURES

In much the same way that coding theorists relay on Shannon's channel-capacity theorem to guide their search for low p(e) codes, researchers on implementation of Viterbi algorithm rely on area-time complexity in order to establish a lower bounds on the performance on any VLSI implementation of the Viterbi algorithm. In fact, the model of VLSI computation due to Thompson [35] can be adopted for performance evaluation of Viterbi algorithm VLSI implementations.

There is an apparent area-time tradeoff available to the designer. Fast architectures which support high baud rates tend to have large chip areas, while slow architectures can be quite small in area. It can be shown [15] that if a VLSI design with area A executes the Viterbi design with alphabet size m and memory v where v is the base 2 logarithm of the number of states in the trellis structure, in symbol

interval T, then the (area)*(baud rate)$^{-2}$ = AT^2 product is :

$$AT^2 = \Omega(m^{2v}) \qquad (6.1)$$

Where a function f(n) is Ω (g(n)) if there exist positive constant k and n_0 such that $f(n) \geq k\,g(n)$ for $n > n_0$. It has also been shown [14] that for any VLSI implementation of the Viterbi algorithm, the energy consumption during each symbol interval (i.e. power) is given by $AT^2 = \Omega(m^{\frac{3v}{2}})$ where AT can be interpreted as the reciprocal of throughput per unit area. Hence, an architecture which is optimized to achieve this lower bound can claim to have made best use of the chip area, and accordingly its performance is superior.

Establishing the AT lower bound for the Viterbi algorithm is important in the sense that it provides necessary conditions on the relationship between certain parameters of a Viterbi algorithm VLSI circuit such as, between baud rate, chip area, algorithm memory length v, power dissipation, etc.

	LV7026 Dec.	Normal Dec.	Systolic Dec.
SPEED	2 Mbits/s	2400 bits/s	12 Mbits/s
AREA (m=7)	356 ICs	_____	9.4 cm**2
AREA (m=10)	_____	< 10 cm**2	98 cm**2

Table 6.2: *Area* and *speed of the three decoders*

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 GENERAL CONCLUSIONS

In this thesis a new systolic algorithm for a Viterbi decoder has been developed. The new systolic algorithm provides simple, regular and modular layouts. A design for a systolic decoder has also been developed using CMOS VLSI technology. The designed decoder provides fast and high throughput rates with minimum chip area and power dissipation.

The systolic decoder developed in this thesis successfully fulfill the performance criterion of simple basic cell design, high array efficiency and high allowable data rates. The comparison made between the systolic decoder and other decoders has explicitly shown how powerful the design of the systolic decoder is. High speed, low power dissipation, and small area occupied of the systolic decoder has made it best candidate for real-time applications.

The contribution of this research can be summarized in the

following:

1) Development of new systolic algorithm for decoding convolutional codes using Viterbi decoding.

2) Development of CMOS VLSI design for the systolic Viterbi decoder.

3) Application of special techniques to enhance the performance of the array cells such as two phase clocking and latch removing.

4) Development of a methodology for designing any k/n rate systolic decoder.

7.2 FUTURE WORK

1) For completeness of chip design fault tolerance study for the systolic decoder can be made. The approach of algorithm-based fault tolerance [1] may be used to provide highly reliable systolic decoder chip. Fault tolerance study is particularly important if the design is to be used in space crafts or in satellite communications where reliability is essential.

2) Design of TTL cells for systolic decoders can be made if faster decoders are needed. TTL systolic decoders might find its applications in environments where extremely high data rates are required and where high power dissipation and large system area

is tolerable. Also TTL systolic decoders might be needed to enhance the performance of k/n rate convolutional codes where k>1.

3) A study for designing soft decision systolic decoders can be carried out to provide decoders with better decoding cababilities. It has been claimed [36] that soft decision does not increase the hardware complexity of any Viterbi decoder by more than 20%.

REFERENCES

[1] J.A. Abraham and others, "Fault tolerance techniques for systolic arrays," Computer, vol. 20, No. 7, pp. 45-49, July 1987.

[2] V.K. Bhargava, D. Haccoun, R. Matyas, P. Nuspl, Digital Communications by Satellites, John wiley & Sons, New York 1981.

[3] J.B. Cain, G.C. Clark, JR., J.M. Geist, "Punctured convolutional codes of rate (n-1)/n and simplified maximum likelihood decoding," IEEE Trans. Inf. Theory, vol. IT-25, No. 1, Jan. 1979.

[4] V. Cappelini, Data Compression and Error Control Techniques with Applications, Academic Press, London 1985.

[5] G.C. Clark and J. Cain, Error-Correction Coding for Digital Communications, Plenum Press 1981.

[6] J. Conan and R. Oliver, "Hardware and software implementation of the Viterbi decoding algorithm for convolutional codes," Proc. ISMM '76., Nov. 1976, Canada.

[7] B.L. Drake and others, "SLAPP: A systolic linear algebra

parallel processor," Computer, vol. 20, No. 7, pp. 45-49, July 1987.

[8] R.D. Dulk and J.J. Stuyt, "A versatile CMOS rate multiplier/variable divider," IEEE Journal of Solid-State Circuits, vol. SC-18, No.3, pp. 267-272, June 1983.

[9] P. Elias, " Coding for noisy channels,"in 1955 IRE Nat. Conv. Rec., Vol 3, pt.4, pp. 37-46.

[10] G.D. Forney, Jr., "Convolutional codes I: Algebraic structure," IEEE Trans. Inform. Theory, vol. IT-16, pp. 720-738, Nov. 1970.

[11] G.D. Forney, Jr., "The Viterbi algorithm," IEEE Proc., vol. 61, No.3, pp. 268-278, March 1973.

[12] J. Fortes and B. Wah, "Systolic arrays: from concept to implementation," Computer, vol. 20, No.7, pp. 12-17, July 1987.

[13] M.J Foster and H.T. Kung, "The design of special purpose VLSI chips," Computer, 1980, pp. 26 -40

[14] N. Frenette, P. McLane, L. Peppard, F. Cotter, " Implementation of a Viterbi processor for a digital communications system with a timedispersive channel," IEEE Journal on Selected Areas in Common., vol. SAC-4, No.1,

pp. 160-167, Jan. 1986.

[15] P.G. Gulak and E. Shwedyk, "VLSI structures for Viterbi Receivers: part I - General theory and applications," IEEE Journal on Selected Areas in Commun, vol. SAC - 4, N0.1, pp. 142-154, Jan 1986.

[16] A. Guyot, A. Jerrage and J. Raymond, "Language Universitaire Conception de Circuits Integres pur 1' Enseighement," IMAG report.

[17] J.A. Heller and I.M. Jacobs, "Viterbi decoding for satellite and space communication," IEEE Trans. Commun. Technol., vol. COM-19, pp. 835-847, Oct. 1971.

[18] F.J. Hill and G.R. Peterson, Digital Systems: Hardware Organization and Design, John wiley & Sons, New York, 1973. (2nd edition published: 1978).

[19] H.T. Kung and C.E. Leiserson, "Systolic Arrays," Sparse Matrix Proc. 1978/1979, Academic Press, Orlando, Fla., pp. 256-282.

[20] H.T. Kung, "Let's design algorithms for VLSI systems," Proc. Caltech conf. on VLSI, pp. 65-90, Jan. 1979.

[21] H.T. Kung, "Why systolic architectures?," Computer, pp.37-46, Jan. 1982.

[22] S.Y. Kung, S.C. Lo, S.N. Jean, J.N. Hwang, "Wavefront array processors - Concept to implementation," Computer, vol. 20, No. 7, pp. 18-33, July 1987.

[23] G.J. Li and B.W. Wah, "The design of optimal systolic arrays," IEEE Trans. on Computers, pp. 66-77, Jan. 1985.

[24] M.M. Mano, Digital Logic and Computer Design, Prentice-Hall, New Jersey 1979.

[25] C. Mead and L. Conway, Introduction to VLSI systems, Addison-Welsey, Reading Mass., 1980.

[26] A. Mukherjee, Introduction to nMos and CMOS VLSI systems, Prentice-Hall, New Jersey 1986.

[27] J.J Navarro, J.M. Llaberio, M. Valero, "Partitioning: An essential step in mapping algorithms into systolic array processors," Computer, vol.20, No.7, pp. 77-89, July 1987.

[28] J.K. Omura, "On the Viterbi decoding algorithm," IEEE Tran. Inf. Theory, vol. IT-15, pp. 177-179, Jan. 1969.

[29] H.F. Rashvand, "Implementation of microprocessors in trellis decoding of convolutional codes," Electronic letters, vol18, No. 3, pp. 121-123, Feb. 1982.

[30] C. Roos, "On the structure of convolutional and cyclic

convolutional codes," IEEE Tran. Inf. Theory, vol. IT-25, No. 6, pp. 676-684, Nov. 1979.

[31] Sadiq M. Sait, Ali F. Damati, Mushfiqur Rahman, "Systolic architecture design for decoding convolutional codes using Viterbi algorithm," Proc. ISMM '88., June 1988, Spain.

[32] Sadiq M. Sait, "VLSI Mask Generation from Register Transfer Level Description : An Automated Approach,". Ph.D. Dissertation, Nov.1986, University of Petroleum and Minerals.

[33] S.M. Said and K.R. Dimond, "Real-time implementation of the Viterbi decoding algorithm on a high-performance microprocessor," Microprocessors and Microsystems, vol. 10, No.1, pp. 11-16, Jan/Feb.1986

[34] C.C. Thompson, "A complexity theory for VLSI," Ph.D. dissertation, Dept. Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, 1980.

[35] A.J. Viterbi, "Error bounds for convolutional codes and an assymptotically optimum decoding algorithm," IEEE Trans. Inf. Theory, vol. IT-13, pp. 260-269, April 1967.

[36] A.J. Viterbi, "Convolutional codes and their performance in communication systems," IEEE Trans. Commun. Technol., vol. COM-19, pp. 751-772, Oct. 1971.

[37] A.J. Viterbi and J.K. Omura, Principles of Digital Communications and Coding, McGraw-Hill 1979.

Appendix A

UAHPL PROGRAM LISTING

```
*************************************************
*                                               *
* Program listing for a 1/2 rate systolic       *
* decoder where the constraint length is 3      *
*                                               *
*************************************************

MODULE  :PROC1.
  MEMORY  : REG1{4};REG2{4}.
  MEMORY  : FLG1;FLG2.
  EXINPUT : IN{2}.
  EXINPUT : CLOCK;RESET.
  BUSES   : G1 ;G2 ;G3.
  BUSES   : C{3};D{3};PREG1{4};PREG2{4}.
  BUSES   : M1{4};M2{4}.
  BUSES   : IMIN12{4}.
  EXBUSES : MIN12{4};MIN56{4}.
  EXBUSES : F11;F12.
  EXBUSES : PATH11;PATH15.
  CLUNITS : COM{3};ADD{4}.
BODY
SEQUENCE  : CLOCK.
1       PREG1 = \0,0,0,0\;
        FLG1<=\1\;
        REG1 <= ADD(2$0,G1,G2;PREG1).
2       C  = COM(REG1;REG2);
        IMIN12= (C{2}&REG1 + ¬C{2}&REG2);
        PREG1 = MIN12;
        PREG2 = MIN56;
        M1 = ADD(2$0,G1,G2;PREG1);
        M2 = ADD(2$0,G3,G2;PREG2);
        D  = COM(M1;M2);
        FLG1= D{2};
        FLG2= ¬D{2};
        F11 <= FLG1;
        F12 <= FLG2;
        REG1  <= M1;
        REG2  <= M2;
        => 2.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
MIN12 = IMIN12;
F11= FLG1; F12= FLG2;
PATH11=FLG1;PATH15=FLG2;

"Hamming Distance Calculations"
```

```
G1 = IN{0} & IN{1};
G2 = IN{0} @ IN{1};
G3 = ¬IN{0} & ¬IN{1}.
END.

MODULE  :PROC2.
  MEMORY   : REG3{4};REG4{4}.
  MEMORY   : FLG1;FLG2.
  EXINPUT  : IN{2}.
  EXINPUT  : CLOCK;RESET.
  BUSES    : G1 ;G2 ;G3 ;PREG1{4};PREG2{4}.
  BUSES    : C{3};D{3}.
  BUSES    : M3{4};M4{4}.
  BUSES    : IMIN2{4}.
  EXBUSES  : MIN12{4};MIN34{4};MIN56{4}.
  EXBUSES  : F13;F14.
  EXBUSES  : PATH12;PATH16.
  CLUNITS  : COM{3};ADD{4}.
BODY
SEQUENCE   : CLOCK.
1       PREG1 = \0,0,0,0\;
        FLG1<=\1\;
        REG3 <= ADD(2$0,G1,G2;PREG1).
2       C  = COM(REG3;REG4);
        IMIN2= (C{2}&REG3 + ¬C{2}&REG4);
        PREG1 = MIN12;
        PREG2 = MIN56;
        M3 = ADD(2$0,G3,G2;PREG1);
        M4 = ADD(2$0,G1,G2;PREG2);
        D  = COM(M3;M4);
        FLG1= D{2};
        FLG2= ¬D{2};
        REG3  <= M3;
        REG4  <= M4;
        =>  2.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
MIN34 = IMIN2;
F13= FLG1; F14= FLG2;
PATH12=FLG1;PATH16=FLG2;

"Hamming Distance Calculations"

G1 = IN{0} & IN{1};
G2 = IN{0} @ IN{1};
G3 = ¬IN{0} & ¬IN{1}.
END.
MODULE  :PROC3.
```

```
  MEMORY   : REG5{4};REG6{4}.
  MEMORY   : FLG1;FLG2.
  EXINPUT  : IN{2}.
  EXINPUT  : CLOCK;RESET.
  BUSES    : G4 ;G5 ;G6 ;PREG1{4};PREG2{4}.
  BUSES    : C{3};D{3}.
  BUSES    : M5{4};M6{4}.
  BUSES    : IMIN3{4}.
  EXBUSES  : MIN34{4};MIN56{4};MIN78{4}.
  EXBUSES  : F15;F16.
  EXBUSES  : PATH13;PATH17.
  CLUNITS  : COM{3};ADD{4}.
BODY
SEQUENCE   : CLOCK.
1       MIN56 = \0,0,1,1\.
2       MIN56 =\0,0,1,1\;
        PREG1    =MIN34;
        FLG1 <=\1\;
        REG5 <= ADD(2$0,G4,G5;PREG1).
3       C  = COM(REG5;REG6);
        IMIN3= (C{2}&REG5 + ¬C{2}&REG6);
        PREG1 = MIN34;
        PREG2 = MIN78;
        M5 = ADD(2$0,G4,G5;PREG1);
        M6 = ADD(2$0,G6,G5;PREG2);
        D  = COM(M5;M6);
        FLG1= D{2};
        FLG2= ¬D{2};
        REG5  <= M5;
        REG6  <= M6;
        => 3.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
MIN56 = IMIN3;
F15= FLG1; F16= FLG2;
PATH13=FLG1;PATH17=FLG2;

"Hamming Distance Calculations"

G4 = IN{0} & ¬IN{1};
G5 = ¬IN{0} & ¬IN{1} + IN{0} & IN{1};
G6 = ¬IN{0} & IN{1}.
END.

MODULE  :PROC4.
  MEMORY   : REG7{4};REG8{4}.
  MEMORY   : FLG1;FLG2.
  EXINPUT  : IN{2}.
```

```
  EXINPUT : CLOCK;RESET.
  BUSES   : G4 ;G5 ;G6 ;PREG1{4};PREG2{4}.
  BUSES   : C{3};D{3}.
  BUSES   : M7{4};M8{4}.
  BUSES   : IMIN4{4}.
  EXBUSES : MIN34{4};MIN78{4}.
  EXBUSES : F17;F18.
  EXBUSES : PATH14;PATH18.
  CLUNITS : COM{3};ADD{4}.
BODY
SEQUENCE  : CLOCK.
1       MIN78 = \0,0,1,1\.
2       MIN78 = \0,0,1,1\;
        PREG1   =MIN34;
        FLG1 <=\1\;
        REG7 <= ADD(2$0,G6,G5;PREG1).
3       C  = COM(REG7;REG8);
        IMIN4= (C{2}&REG7 + ¬C{2}&REG8);
        PREG1 = MIN34;
        PREG2 = IMIN4;
        M7 = ADD(2$0,G6,G5;PREG1);
        M8 = ADD(2$0,G4,G5;PREG2);
        D  = COM(M7;M8);
        FLG1= D{2};
        FLG2= ¬D{2};
        REG7  <= M7;
        REG8  <= M8;
        =>  3.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
MIN78 = IMIN4;
F17= FLG1; F18= FLG2;
PATH14=FLG1;PATH18=FLG2;

"Hamming Distance Calculations"

G4 = IN{0} & ¬IN{1};
G5 = ¬IN{0} & ¬IN{1} + IN{0} & IN{1};
G6 = ¬IN{0} & IN{1}.
END.
MODULE    : P11.
  MEMORY  : FLG11;FLG12.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F11;F12.
  EXBUSES : F21;F22.
  EXBUSES : PATH11;PATH12.
  EXBUSES : PATH21;PATH25.
```

```
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F11&(PATH11+PATH12);
      UPFLG2=F12&(PATH11+PATH12);
      FLG11  <=UPFLG1;
      FLG12  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH21 = UPFLG1;
      PATH25 = UPFLG2;
      F21   =FLG11;
      F22   =FLG12.
END.
MODULE    : P12.
  MEMORY  : FLG13;FLG14.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F13;F14.
  EXBUSES : F23;F24.
  EXBUSES : PATH13;PATH14.
  EXBUSES : PATH22;PATH26.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F13&(PATH13+PATH14);
      UPFLG2=F14&(PATH13+PATH14);
      FLG13  <=UPFLG1;
      FLG14  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH22 = UPFLG1;
      PATH26 = UPFLG2;
      F23   =FLG13;
      F24   =FLG14.
END.
MODULE    : P13.
  MEMORY  : FLG15;FLG16.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F15;F16.
  EXBUSES : F25;F26.
  EXBUSES : PATH15;PATH16.
  EXBUSES : PATH23;PATH27.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F15&(PATH15+PATH16);
      UPFLG2=F16&(PATH15+PATH16);
```

```
      FLG15  <=UPFLG1;
      FLG16  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH23 = UPFLG1;
      PATH27 = UPFLG2;
      F25   =FLG15;
      F26   =FLG16.
END.
MODULE    : P14.
  MEMORY  : FLG17;FLG18.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F17;F18.
  EXBUSES : F27;F28.
  EXBUSES : PATH17;PATH18.
  EXBUSES : PATH24;PATH28.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F17&(PATH17+PATH18);
      UPFLG2=F18&(PATH17+PATH18);
      FLG17  <=UPFLG1;
      FLG18  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH24 = UPFLG1;
      PATH28 = UPFLG2;
      F27   =FLG17;
      F28   =FLG18.
END.
MODULE    : P21.
  MEMORY  : FLG21;FLG22.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F21;F22.
  EXBUSES : F31;F32.
  EXBUSES : PATH21;PATH22.
  EXBUSES : PATH31;PATH35.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F21&(PATH21+PATH22);
      UPFLG2=F22&(PATH21+PATH22);
      FLG21  <=UPFLG1;
      FLG22  <=UPFLG2;
       => 1.
ENDSEQUENCE
```

```
CONTROLRESET (RESET)/(1);
      PATH31 = UPFLG1;
      PATH35 = UPFLG2;
      F31   =FLG21;
      F32   =FLG22.
END.
MODULE    : P22.
  MEMORY  : FLG23;FLG24.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F23;F24.
  EXBUSES : F33;F34.
  EXBUSES : PATH23;PATH24.
  EXBUSES : PATH32;PATH36.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F23&(PATH23+PATH24);
      UPFLG2=F24&(PATH23+PATH24);
      FLG23  <=UPFLG1;
      FLG24  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH32 = UPFLG1;
      PATH36 = UPFLG2;
      F33   =FLG23;
      F34   =FLG24.
END.
MODULE    : P23.
  MEMORY  : FLG25;FLG26.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F25;F26.
  EXBUSES : F35;F36.
  EXBUSES : PATH25;PATH26.
  EXBUSES : PATH33;PATH37.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F25&(PATH25+PATH26);
      UPFLG2=F26&(PATH25+PATH26);
      FLG25  <=UPFLG1;
      FLG26  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH33 = UPFLG1;
      PATH37 = UPFLG2;
      F35   =FLG25;
```

```
      F36   =FLG26.
END.
MODULE    : P24.
  MEMORY  : FLG27;FLG28.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F27;F28.
  EXBUSES : F37;F38.
  EXBUSES : PATH27;PATH28.
  EXBUSES : PATH34;PATH38.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F27&(PATH27+PATH28);
      UPFLG2=F28&(PATH27+PATH28);
      FLG27  <=UPFLG1;
      FLG28  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH34 = UPFLG1;
      PATH38 = UPFLG2;
      F37   =FLG27;
      F38   =FLG28.
END.
MODULE    : P31.
  MEMORY  : FLG31;FLG32.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F31;F32.
  EXBUSES : F41;F42.
  EXBUSES : PATH31;PATH32.
  EXBUSES : PATH41;PATH45.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F31&(PATH31+PATH32);
      UPFLG2=F32&(PATH31+PATH32);
      FLG31  <=UPFLG1;
      FLG32  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH41 = UPFLG1;
      PATH45 = UPFLG2;
      F41   =FLG31;
      F42   =FLG32.
END.
MODULE    : P32.
  MEMORY  : FLG33;FLG34.
```

```
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F33;F34.
  EXBUSES : F43;F44.
  EXBUSES : PATH33;PATH34.
  EXBUSES : PATH42;PATH46.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F33&(PATH33+PATH34);
      UPFLG2=F34&(PATH33+PATH34);
      FLG33  <=UPFLG1;
      FLG34  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH42 = UPFLG1;
      PATH46 = UPFLG2;
      F43   =FLG33;
      F44   =FLG34.
END.
MODULE    : P33.
  MEMORY  : FLG35;FLG36.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F35;F36.
  EXBUSES : F45;F46.
  EXBUSES : PATH35;PATH36.
  EXBUSES : PATH43;PATH47.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F35&(PATH35+PATH36);
      UPFLG2=F36&(PATH35+PATH36);
      FLG35  <=UPFLG1;
      FLG36  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH43 = UPFLG1;
      PATH47 = UPFLG2;
      F45   =FLG35;
      F46   =FLG36.
END.
MODULE    : P34.
  MEMORY  : FLG37;FLG38.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F37;F38.
  EXBUSES : F47;F48.
```

```
  EXBUSES : PATH37;PATH38.
  EXBUSES : PATH44;PATH48.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F37&(PATH37+PATH38);
      UPFLG2=F38&(PATH37+PATH38);
      FLG37  <=UPFLG1;
      FLG38  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH44 = UPFLG1;
      PATH48 = UPFLG2;
      F47   =FLG37;
      F48   =FLG38.
END.
MODULE    : P41.
  MEMORY  : FLG41;FLG42.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F41;F42.
  EXBUSES : F51;F52.
  EXBUSES : PATH41;PATH42.
  EXBUSES : PATH51;PATH55.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F41&(PATH41+PATH42);
      UPFLG2=F42&(PATH41+PATH42);
      FLG41  <=UPFLG1;
      FLG42  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH51 = UPFLG1;
      PATH55 = UPFLG2;
      F51   =FLG41;
      F52   =FLG42.
END.
MODULE    : P42.
  MEMORY  : FLG43;FLG44.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F43;F44.
  EXBUSES : F53;F54.
  EXBUSES : PATH43;PATH44.
  EXBUSES : PATH52;PATH56.
BODY
SEQUENCE  : CLOCK.
```

```
1     UPFLG1=F43&(PATH43+PATH44);
      UPFLG2=F44&(PATH43+PATH44);
      FLG43  <=UPFLG1;
      FLG44  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH52 = UPFLG1;
      PATH56 = UPFLG2;
      F53   =FLG43;
      F54   =FLG44.
END.
MODULE    : P43.
  MEMORY  : FLG45;FLG46.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F45;F46.
  EXBUSES : F55;F56.
  EXBUSES : PATH45;PATH46.
  EXBUSES : PATH53;PATH57.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F45&(PATH45+PATH46);
      UPFLG2=F46&(PATH45+PATH46);
      FLG45  <=UPFLG1;
      FLG46  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH53 = UPFLG1;
      PATH57 = UPFLG2;
      F55   =FLG45;
      F56   =FLG46.
END.
MODULE    : P44.
  MEMORY  : FLG47;FLG48.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F47;F48.
  EXBUSES : F57;F58.
  EXBUSES : PATH47;PATH48.
  EXBUSES : PATH54;PATH58.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F47&(PATH47+PATH48);
      UPFLG2=F48&(PATH47+PATH48);
      FLG47  <=UPFLG1;
      FLG48  <=UPFLG2;
```

```
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH54 = UPFLG1;
      PATH58 = UPFLG2;
      F57   =FLG47;
      F58   =FLG48.
END.
MODULE    : P51.
  MEMORY  : FLG51;FLG52.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F51;F52.
  EXBUSES : F61;F62.
  EXBUSES : PATH51;PATH52.
  EXBUSES : PATH61;PATH65.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F51&(PATH51+PATH52);
      UPFLG2=F52&(PATH51+PATH52);
      FLG51  <=UPFLG1;
      FLG52  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH61 = UPFLG1;
      PATH65 = UPFLG2;
      F61   =FLG51;
      F62   =FLG52.
END.
MODULE    : P52.
  MEMORY  : FLG53;FLG54.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F53;F54.
  EXBUSES : F63;F64.
  EXBUSES : PATH53;PATH54.
  EXBUSES : PATH62;PATH66.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F53&(PATH53+PATH54);
      UPFLG2=F54&(PATH53+PATH54);
      FLG53  <=UPFLG1;
      FLG54  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH62 = UPFLG1;
```

```
      PATH66 = UPFLG2;
      F63   =FLG53;
      F64   =FLG54.
END.
MODULE    : P53.
  MEMORY  : FLG55;FLG56.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F55;F56.
  EXBUSES : F65;F66.
  EXBUSES : PATH55;PATH56.
  EXBUSES : PATH63;PATH67.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F55&(PATH55+PATH56);
      UPFLG2=F56&(PATH55+PATH56);
      FLG55  <=UPFLG1;
      FLG56  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH63 = UPFLG1;
      PATH67 = UPFLG2;
      F65   =FLG55;
      F66   =FLG56.
END.
MODULE    : P54.
  MEMORY  : FLG57;FLG58.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F57;F58.
  EXBUSES : F67;F68.
  EXBUSES : PATH57;PATH58.
  EXBUSES : PATH64;PATH68.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F57&(PATH57+PATH58);
      UPFLG2=F58&(PATH57+PATH58);
      FLG57  <=UPFLG1;
      FLG58  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH64 = UPFLG1;
      PATH68 = UPFLG2;
      F67   =FLG57;
      F68   =FLG58.
END.
```

```
MODULE     : P61.
  MEMORY   : FLG61;FLG62.
  EXINPUT  : CLOCK;RESET.
  BUSES    : UPFLG1;UPFLG2.
  EXBUSES  : F61;F62.
  EXBUSES  : F71;F72.
  EXBUSES  : PATH61;PATH62.
  EXBUSES  : PATH71;PATH75.
BODY
SEQUENCE   : CLOCK.
1      UPFLG1=F61&(PATH61+PATH62);
       UPFLG2=F62&(PATH61+PATH62);
       FLG61  <=UPFLG1;
       FLG62  <=UPFLG2;
        => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
       PATH71 = UPFLG1;
       PATH75 = UPFLG2;
       F71    =FLG61;
       F72    =FLG62.
END.
MODULE     : P62.
  MEMORY   : FLG63;FLG64.
  EXINPUT  : CLOCK;RESET.
  BUSES    : UPFLG1;UPFLG2.
  EXBUSES  : F63;F64.
  EXBUSES  : F73;F74.
  EXBUSES  : PATH63;PATH64.
  EXBUSES  : PATH72;PATH76.
BODY
SEQUENCE   : CLOCK.
1      UPFLG1=F63&(PATH63+PATH64);
       UPFLG2=F64&(PATH63+PATH64);
       FLG63  <=UPFLG1;
       FLG64  <=UPFLG2;
        => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
       PATH72 = UPFLG1;
       PATH76 = UPFLG2;
       F73    =FLG63;
       F74    =FLG64.
END.
MODULE     : P63.
  MEMORY   : FLG65;FLG66.
  EXINPUT  : CLOCK;RESET.
  BUSES    : UPFLG1;UPFLG2.
```

```
   EXBUSES : F65;F66.
   EXBUSES : F75;F76.
   EXBUSES : PATH65;PATH66.
   EXBUSES : PATH73;PATH77.
 BODY
 SEQUENCE  : CLOCK.
 1      UPFLG1=F65&(PATH65+PATH66);
        UPFLG2=F66&(PATH65+PATH66);
        FLG65  <=UPFLG1;
        FLG66  <=UPFLG2;
         => 1.
 ENDSEQUENCE
 CONTROLRESET (RESET)/(1);
        PATH73 = UPFLG1;
        PATH77 = UPFLG2;
        F75   =FLG65;
        F76   =FLG66.
 END.
 MODULE    : P64.
   MEMORY  : FLG67;FLG68.
   EXINPUT : CLOCK;RESET.
   BUSES   : UPFLG1;UPFLG2.
   EXBUSES : F67;F68.
   EXBUSES : F77;F78.
   EXBUSES : PATH67;PATH68.
   EXBUSES : PATH74;PATH78.
 BODY
 SEQUENCE  : CLOCK.
 1      UPFLG1=F67&(PATH67+PATH68);
        UPFLG2=F68&(PATH67+PATH68);
        FLG67  <=UPFLG1;
        FLG68  <=UPFLG2;
         => 1.
 ENDSEQUENCE
 CONTROLRESET (RESET)/(1);
        PATH74 = UPFLG1;
        PATH78 = UPFLG2;
        F77   =FLG67;
        F78   =FLG68.
 END.
 MODULE    : P71.
   MEMORY  : FLG71;FLG72.
   EXINPUT : CLOCK;RESET.
   BUSES   : UPFLG1;UPFLG2.
   EXBUSES : F71;F72.
   EXBUSES : F81;F82.
   EXBUSES : PATH71;PATH72.
   EXBUSES : PATH81;PATH85.
```

```
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F71&(PATH71+PATH72);
      UPFLG2=F72&(PATH71+PATH72);
      FLG71  <=UPFLG1;
      FLG72  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH81 = UPFLG1;
      PATH85 = UPFLG2;
      F81   =FLG71;
      F82   =FLG72.
END.
MODULE    : P72.
  MEMORY  : FLG73;FLG74.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F73;F74.
  EXBUSES : F83;F84.
  EXBUSES : PATH73;PATH74.
  EXBUSES : PATH82;PATH86.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F73&(PATH73+PATH74);
      UPFLG2=F74&(PATH73+PATH74);
      FLG73  <=UPFLG1;
      FLG74  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH82 = UPFLG1;
      PATH86 = UPFLG2;
      F83   =FLG73;
      F84   =FLG74.
END.
MODULE    : P73.
  MEMORY  : FLG75;FLG76.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F75;F76.
  EXBUSES : F85;F86.
  EXBUSES : PATH75;PATH76.
  EXBUSES : PATH83;PATH87.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F75&(PATH75+PATH76);
      UPFLG2=F76&(PATH75+PATH76);
```

```
      FLG75  <=UPFLG1;
      FLG76  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH83 = UPFLG1;
      PATH87 = UPFLG2;
      F85   =FLG75;
      F86   =FLG76.
END.
MODULE    : P74.
  MEMORY  : FLG77;FLG78.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F77;F78.
  EXBUSES : F87;F88.
  EXBUSES : PATH77;PATH78.
  EXBUSES : PATH84;PATH88.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F77&(PATH77+PATH78);
      UPFLG2=F78&(PATH77+PATH78);
      FLG77  <=UPFLG1;
      FLG78  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH84 = UPFLG1;
      PATH88 = UPFLG2;
      F87   =FLG77;
      F88   =FLG78.
END.
MODULE    : P81.
  MEMORY  : FLG81;FLG82.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F81;F82.
  EXBUSES : F91;F92.
  EXBUSES : PATH81;PATH82.
  EXBUSES : PATH91;PATH95.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F81&(PATH81+PATH82);
      UPFLG2=F82&(PATH81+PATH82);
      FLG81  <=UPFLG1;
      FLG82  <=UPFLG2;
       => 1.
ENDSEQUENCE
```

```
CONTROLRESET (RESET)/(1);
      PATH91 = UPFLG1;
      PATH95 = UPFLG2;
      F91   =FLG81;
      F92   =FLG82.
END.
MODULE    : P82.
  MEMORY  : FLG83;FLG84.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F83;F84.
  EXBUSES : F93;F94.
  EXBUSES : PATH83;PATH84.
  EXBUSES : PATH92;PATH96.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F83&(PATH83+PATH84);
      UPFLG2=F84&(PATH83+PATH84);
      FLG83  <=UPFLG1;
      FLG84  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH92 = UPFLG1;
      PATH96 = UPFLG2;
      F93   =FLG83;
      F94   =FLG84.
END.
MODULE    : P83.
  MEMORY  : FLG85;FLG86.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F85;F86.
  EXBUSES : F95;F96.
  EXBUSES : PATH85;PATH86.
  EXBUSES : PATH93;PATH97.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F85&(PATH85+PATH86);
      UPFLG2=F86&(PATH85+PATH86);
      FLG85  <=UPFLG1;
      FLG86  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH93 = UPFLG1;
      PATH97 = UPFLG2;
      F95   =FLG85;
```

```
      F96   =FLG86.
END.
MODULE    : P84.
  MEMORY  : FLG87;FLG88.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F87;F88.
  EXBUSES : F97;F98.
  EXBUSES : PATH87;PATH88.
  EXBUSES : PATH94;PATH98.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F87&(PATH87+PATH88);
      UPFLG2=F88&(PATH87+PATH88);
      FLG87  <=UPFLG1;
      FLG88  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH94 = UPFLG1;
      PATH98 = UPFLG2;
      F97   =FLG87;
      F98   =FLG88.
END.
MODULE    : P91.
  MEMORY  : FLG91;FLG92.
  EXINPUT : CLOCK;RESET.
  BUSES   : UPFLG1;UPFLG2.
  EXBUSES : F91;F92.
  EXBUSES : FA1;FA2.
  EXBUSES : PATH91;PATH92.
  EXBUSES : PATHA1;PATHA5.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F91&(PATH91+PATH92);
      UPFLG2=F92&(PATH91+PATH92);
      FLG91  <=UPFLG1;
      FLG92  <=UPFLG2;
       => 1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATHA1 = UPFLG1;
      PATHA5 = UPFLG2;
      FA1   =FLG91;
      FA2   =FLG92.
END.
MODULE    : P92.
  MEMORY  : FLG93;FLG94.
```

```
   EXINPUT : CLOCK;RESET.
   BUSES   : UPFLG1;UPFLG2.
   EXBUSES : F93;F94.
   EXBUSES : FA3;FA4.
   EXBUSES : PATH93;PATH94.
   EXBUSES : PATHA2;PATHA6.
 BODY
 SEQUENCE  : CLOCK.
 1     UPFLG1=F93&(PATH93+PATH94);
       UPFLG2=F94&(PATH93+PATH94);
       FLG93  <=UPFLG1;
       FLG94  <=UPFLG2;
        => 1.
 ENDSEQUENCE
 CONTROLRESET (RESET)/(1);
       PATHA2 = UPFLG1;
       PATHA6 = UPFLG2;
       FA3   =FLG93;
       FA4   =FLG94.
 END.
 MODULE    : P93.
   MEMORY  : FLG95;FLG96.
   EXINPUT : CLOCK;RESET.
   BUSES   : UPFLG1;UPFLG2.
   EXBUSES : F95;F96.
   EXBUSES : F105;F106.
   EXBUSES : PATH95;PATH96.
   EXBUSES : PATH103;PATH107.
 BODY
 SEQUENCE  : CLOCK.
 1     UPFLG1=F95&(PATH95+PATH96);
       UPFLG2=F96&(PATH95+PATH96);
       FLG95  <=UPFLG1;
       FLG96  <=UPFLG2;
        => 1.
 ENDSEQUENCE
 CONTROLRESET (RESET)/(1);
       PATH103 = UPFLG1;
       PATH107 = UPFLG2;
       F105   =FLG95;
       F106   =FLG96.
 END.
 MODULE    : P94.
   MEMORY  : FLG97;FLG98.
   EXINPUT : CLOCK;RESET.
   BUSES   : UPFLG1;UPFLG2.
   EXBUSES : F97;F98.
   EXBUSES : F107;F108.
```

```
  EXBUSES : PATH97;PATH98.
  EXBUSES : PATH104;PATH108.
BODY
SEQUENCE  : CLOCK.
1     UPFLG1=F97&(PATH97+PATH98);
      UPFLG2=F98&(PATH97+PATH98);
      FLG97  <=UPFLG1;
      FLG98  <=UPFLG2;
       =>  1.
ENDSEQUENCE
CONTROLRESET (RESET)/(1);
      PATH104 = UPFLG1;
      PATH108 = UPFLG2;
      F107   =FLG97;
      F108   =FLG98.
END.
```

Appendix B

SPICE PROGRAM LISTING

Four-Bit Adder

```
*****************************************************
*
*   Simulation of a 4-bit adder
*     First input  : A4 A3 A2 A1
*     Second input : B4 B3 B2 B1
*     Output       : SUM4 SUM3 SUM2 SUM1
*     Carry out    : COUT
*
*****************************************************

 VDD 100 0 DC 5
 VP1 1 0 PULSE 0 5 640NS 0NS 0NS 640NS 1280NS
 VP2 2 0 PULSE 0 5 320NS 0NS 0NS 320NS 640NS
 VP3 3 0 PULSE 0 5 160NS 0NS 0NS 160NS 320NS
 VP4 4 0 PULSE 0 5 080NS 0NS 0NS 080NS 160NS
 VP11 11 0 PULSE 0 5 40NS 0NS 0NS 40NS 80NS
 VP22 22 0 PULSE 0 5 20NS 0NS 0NS 20NS 40NS
 VP33 33 0 PULSE 0 5 10NS 0NS 0NS 10NS 20NS
 VP44 44 0 PULSE 0 5 05NS 0NS 0NS 05NS 10NS
 .TRAN 1NS 1280NS
 .OPTIONS ITL4=40 ITL5=9000
 .OPTIONS RELTOL=.005
 *
 * CALL 4-BIT ADDER SUBROUTINE
 *
 X99 1 2 3 4 11 22 33 44 6 7 8 9 10 100 0  ADDER
 * A1/A2/A3/A4/B1/B2/B3/B4/SUM1/SUM2/SUM3/SUM4/COUT/VDD/VSS
 .PLOT TRAN V(1)
 .PLOT TRAN V(2)
 .PLOT TRAN V(3)
 .PLOT TRAN V(4)
 .PLOT TRAN V(11)
 .PLOT TRAN V(22)
 .PLOT TRAN V(33)
 .PLOT TRAN V(44)
 .PLOT TRAN V(6)
 .PLOT TRAN V(7)
 .PLOT TRAN V(8)
 .PLOT TRAN V(9)
 .PLOT TRAN V(10)
 *
 * 4-BIT ADDER
 *
 .SUBCKT ADDER 1 11 21 31 2 12 22 32 5 15 25 35 34 33 44
 * A1/A2/A3/A4/B1/B2/B3/B4/SUM1/SUM2/SUM3/SUM4/COUT/VDD/VSS
 *
```

```
X11 1  2  3  4  5  33 44 ADDER1
X12 11 12 4  14 15 33 44 ADDER1
X13 21 22 14 24 25 33 44 ADDER1
X14 31 32 24 34 35 33 44 ADDER1
.ENDS
*
* 1-BIT ADDER
*
.SUBCKT ADDER1 1 2 3 4 5 33 44
* IN1/IN2/CIN/COUT/SUM/VDD/VSS
*
X1  1  10 33 44  INV
X2  10 2  2  1  6  33 44  PPNN
X3  6  2  1  10 33 44 TGATE
X4  2  4  6  11 33 44 TGATE
X5  4  3  11 6  33 44 TGATE
X6  3  5  6  11 33 44 TGATE
X7  11 3  3  6  5  33 44 PPNN
X8  6  11 33 44 INV
C1  1 0 0.093PF
C2  10 0 0.093PF
C3  2  0 0.093PF
C4  3  0 0.093PF
.ENDS
*
* INVARTER
*
.SUBCKT INV 1 2 3 4
* IN/OUT/VDD/VSS
*
M1 3 1 2 3 P L=2U W=4U
M2 2 1 4 4 N L=2U W=4U
.ENDS
*
* TRANSMISSION GATE
*
.SUBCKT TGATE 1 2 3 4 5 6
* IN/OUT/PMOS/NMOS/VDD/DSS
M1 1 3 2 5 P L=2U W=4U
M2 1 4 2 6 N L=2U W=4U
.ENDS
*
* PPNN CIRCUIT
*
.SUBCKT PPNN 1 2 3 4 5 8 9
* INP1/INP2/INN1/INN2/OUT/VDD/VSS
M1 8 1 6 8 P L=2U W=4U
M2 6 2 5 8 P L=2U W=4U
```

```
M3 5 3 7 9 N L=2U W=4U
M4 7 4 9 9 P L=2U W=4U
.ENDS
*
.MODEL P PMOS
.MODEL N NMOS
.END
```

Four-Bit Magnitude Comparator

```
**********************************************************
*                                                        *
* Simulation of 4-bit magnitude comparator where         *
* A3 A2 A1 A0 and B3 B2 B1 B0 are compared.              *
*                                                        *
**********************************************************

VDD 100 0 DC 5
*
* Clock pulses for A3 A2 A1 A0 and B3 B2 B1 B0
*
VP1 1 0 PULSE 0 5 640NS 0NS 0NS 640NS 1280NS
VP3 3 0 PULSE 0 5 320NS 0NS 0NS 320NS 640NS
VP5 5 0 PULSE 0 5 160NS 0NS 0NS 160NS 320NS
VP7 7 0 PULSE 0 5 080NS 0NS 0NS 080NS 160NS
VP2 2 0 PULSE 0 5 040NS 0NS 0NS 040NS 80NS
VP4 4 0 PULSE 0 5 020NS 0NS 0NS 020NS 40NS
VP6 6 0 PULSE 0 5 010NS 0NS 0NS 010NS 20NS
VP8 8 0 PULSE 0 5 005NS 0NS 0NS 005NS 10NS
.TRAN 1NS 1280NS
.OPTIONS ITL4=100 ITL5=50000
.OPTIONS RELTOL=.005
*
* CALL 4-BIT COMPARATOR SUBROUTINE
*
X99 1 3 5 7 2 4 6 8 80 90 99 100 0 CMPR
* A3/A2/A1/A0/B3/B2/B1/B0/A=B/A<B/A=<B/VDD/VSS
*
.PLOT TRAN V(1) V(3) V(5) V(7)
.PLOT TRAN V(2) V(4) V(6) V(8)
.PLOT TRAN V(80)
.PLOT TRAN V(90)
.PLOT TRAN V(99)
*
* 4-BIT COMPARATOR SUBROUTINE
*
.SUBCKT CMPR 1 3 5 7 2 4 6 8 80 90 99 33 44
X1  1  2  9  10 11 33 44  CMPR1
X2  3  4  19 20 21 33 44  CMPR1
X3  5  6  29 30 31 33 44  CMPR1
X4  7  8  39 40 41 33 44  CMPR1
*
* ANDING
*
X5  11  20  49  33 44 NAND
X6  49  50  33  44 INV
```

```
*
X7  11  21  57  33 44 NAND
X8  57  58  33  44 INV
X9  30  58  59  33 44 NAND
X10 59  60  33  44 INV
*
X11 31  40  67  33 44 NAND
X12 67  68  33  44 INV
X13 58  68  69  33 44 NAND
X14 69  70  33  44 INV
*
X15 41  31  77  33 44 NAND
X16 77  78  33  44 INV
X17 58  78  80  33 44 NAND
*
* ORING, THE OUTPUT DETERMINES  A<B OR NOT
*
X18 10 50 81 33 44 NOR
X19 60 70 82 33 44 NOR
X20 81 83 33 44 INV
X21 82 84 33 44 INV
X22 83 84 89 33 44 NOR
X23 89 90 33 44 INV
*
* DETERMINATION OF A =< B OR NOT
*
X24 80 90 98 33 44 NOR
X25 98 99 33 44 INV
.ENDS
*
.SUBCKT CMPR1 1 2 9 10 11 33 44
* IN1/IN2/NOR2/NOR1/NOR3/VDD/VSS
*
X31 1 3 33 44 INV
X32 2 4 33 44 INV
X33 1  4  10 33 44 NOR
X34 2  3  9  33 44 NOR
X35 9  10 11 33 44 NOR
.ENDS
*
* NAND GATE
*
.SUBCKT NAND 1 2 3 5 6
* IN1/IN2/OUT/VDD/VSS
*
M1 5 1 3 5 P L=2U W=4U
M2 5 2 3 5 P L=2U W=4U
M3 3 1 4 6 N L=2U W=4U
```

```
M4 4 2 6 6 N L=2U W=4U
.ENDS
*
* INVARTER
*
.SUBCKT INV 1 2 3 4
* IN/OUT/VDD/VSS
*
M1 3 1 2 3 P L=2U W=4U
M2 2 1 4 4 N L=2U W=4U
.ENDS
*
* NOR GATE
*
.SUBCKT NOR 4 2 5 1 6
* IN1/IN2/OUT/VDD/VSS
*
M1 1 2 3 1 P L=2U W=4U
M2 3 4 5 1 P L=2U W=4U
M3 5 4 6 6 N L=2U W=4U
M4 5 2 6 6 N L=2U W=4U
.ENDS
.MODEL P PMOS
.MODEL N NMOS
.END
```

Appendix C

LUCIE PROGRAM LISTING

```
*********************************************
*                                           *
*     LUCIE file for a 4-bit adder          *
*                                           *
*********************************************

niv dn,dp,co,po,vi,m2,pa,bo
FIG ADDER4
    figext adder1(20,20)
    figext adder1(350,20)
    figext adder1(680,20)
    figext adder1(1010,20)
FFIG

* Full Adder Routine (ADDER1)

niv dn,dp,co,po,vi,m2,pa,bo
FIG ADDER1
    figext adder(15,15)
    rec(156,17,23,2,po)
    rec(177,13,6,6,po)
    rec(179,15,2,2,co)
    rec(224,13,6,6,po)
    rec(226,15,2,2,co)
    rec(177,13,53,6,m1)
    rec(192,7,2,21,po)
    rec(140,7,54,2,po)
    rec(140,7,2,86,po)
    rec(17,51,8,2,po)
    rec(17,51,2,40,po)
    rec(133,6,2,47,po)
    rec(83,6,2,30,po)
    rec(83,6,52,2,po)
    rec(83,65,2,39,po)
    rec(325,51,15,2,po)
    rec(139,7,147,2,po)
    rec(338,8,2,45,po)
    rec(245,51,2,56,po)
    rec(130,28,16,6,m1)
    rec(168,28,14,6,m1)
    rec(203,28,15,6,m1)
    rec(204,70,13,6,m1)
    rec(168,70,14,6,m1)
    rec(132,70,13,6,m1)
    rec(261,91,74,6,m1)
    rec(169,91,6,6,po)
    rec(171,93,2,2,co)
    rec(0,91,175,6,m1)
```

```
    rec(17,84,9,22,po)
    rec(82,98,9,8,po)
    rec(257,50,6,6,po)
    rec(259,52,2,2,co)
    rec(243,98,10,8,po)
FFIG
```

```
**********************************************
*                                            *
* LUCIE file for a 4-bit magnitude comparator *
*                                            *
**********************************************

niv dn,dp,co,po,vi,m2,pa,bo
FIG CMPR
    figext cmpr1(0,0)
    figext cmpr1(0,140)
    figext cmpr1(0,280)
    figext cmpr1(0,420)
    figext cmpr1(170,6)
    figext cmpr1(170,73)
    figext cmpr1(170,146)
    figext cmpr1(170,213)
    figext cmpr1(170,286)
    figext cmpr1(170,353)
    figext cmpr1(170,426)
    rec (129,452,13,2,po)
    rec (193,408,2,31,po)
    rec (193,483,2,7,po)
    rec (140,488,55,2,po)
    rec (140,452,2,37,po)
    rec (145,451,38,2,po)
    rec (145,379,2,74,po)
    rec (113,379,34,2,po)
    rec (150,379,32,2,po)
    rec (150,312,2,68,po)
    rec (131,312,21,2,po)
    rec (156,312,27,2,po)
    rec (156,239,2,75,po)
    rec (113,239,44,2,po)
    rec (129,172,54,3,po)
    rec (192,138,2,22,po)
    rec (113,138,81,2,o)
    rec (113,99,2,41,po)
    rec (129,32,11,2,po)
    rec (138,32,2,72,po)
    rec (193,269,2,30,po)
    rec (192,130,22,2,po)
    rec (211,130,3,83,po)
    figext pinv (232,427)
    figext pinv (232,354)
    figext pinv (232,287)
    figext pinv (232,214)
    figext pinv (232,145)
```

```
    figext pinv (232,74)
    figext norc (232,4)
    rec (201,453,32,2,po)
    rec (202,380,31,2,po)
    rec (192,211,2,15,po)
    rec (192,211,22,2,po)
    rec (201,240,32,2,po)
    rec (201,313,34,2,po)
    rec (199,99,34,2,po)
    rec (226,171,6,6,po)
    rec (226,171,6,6,m1)
    rec (199,172,33,6,m1)
    rec (228,173,2,2,co)
    figext pinv (295,4)
    figext pinv (295,72)
    figext norc (295,145)
    figext norc (295,427)
    figext pinv (295,353)
    figext norc (295,286)
    figext pinv (295,213)
FFIG

* CMPR1 Routine
niv dn,dp,co,po,vi,m2,pa,bo
FIG CMPR1
    figext pinv(3,7)
    figext pinv(3,74)
    figext norc(41,7)
    figext norc(41,74)
    rec(30,100,29,2,po)
    rec(29,33,31,2,po)
    rec(17,47,2,17,po)
    rec(17,62,49,2,po)
    rec(64,63,2,14,po)
    rec(3,64,73,6,m1)
    rec(70,64,6,6,po)
    rec(3,64,6,6,po)
    rec(74,56,2,14,po)
    rec(64,56,12,2,po)
    rec(72,66,2,2,co)
    rec(17,69,2,17,po)
    rec(3,68,16,2,po)
    rec(5,66,2,2,co)
    figevt norc(90,7)
    rec(80,32,29,2,po)
    rec(113,56,2,45,po)
    rec(80,99,35,2,po)
FFIG
```

```
**************************************************
*                                                *
*       LUCIE file for a NOR gate                *
*                                                *
**************************************************

niv  dn,dp,co,po,m1,cn,cp,vi,m2,pa,bo
FIG  NORA
  *device
    rec (0,0,42,22,cp)
    rec (0,31,42,25,cn)
    rec (3,34,6,16,m1)
    rec (33,13,6,27,m1)
   *contacts
    rec (30,5,2,2,co)
    rec (5,15,2,2,co)
    rec (5,36,2,2,co)
    rec (5,46,2,2,co)
    rec (17,5,2,2,co)
    rec (35,15,2,2,co)
    rec (35,36,2,2,co)
   *diffusion
    rec (3,13,36,6,dn)
    rec (15,3,6,11,dn)
    rec (3,34,36,6,dp)
    rec (28,3,6,6,dp)
    rec (23,2,2,48,po)
    rec (3,44,6,6,dn)
    rec (3,13,36,6,m1)
    rec (17,24,2,25,po)
    rec (9,24,10,2,po)
    rec (9,2,2,24,po)
FFIG
```

```
*************************************************
*                                               *
*      LUCIE file for a NAND gate                *
*                                               *
*************************************************

niv dn,dp,co,po,m1,cn,cp,vi,m2,pa,bo
FIG NAND
  *n device
    rec (0,0,36,26,cp)
    rec (0,31,36,27,cn)
    rec (0,4,35,6,m1)
    rec (3,4,6,16,m1)
    rec (3,4,6,6,dp)
    rec (5,6,2,2,co)
    rec (3,14,6,6,dn)
    rec (5,16,2,2,co)
    rec (9,14,17,4,dn)
    rec (26,14,6,6,dn)
    rec (26,14,6,6,m1)
    rec (28,16,2,2,co)
    rec (11,12,2,46,po)
    rec (22,12,2,19,po)
    rec (3,37,6,6,m1)
    rec (27,37,6,6,m1)
    rec (9,40,19,3,m1)
    rec (29,39,2,2,co)
    rec (5,39,2,2,co)
    rec (3,37,30,6,dp)
    rec (0,46,35,6,m1)
    rec (27,47,6,6,dn)
    rec (29,49,2,2,co)
    rec (15,43,4,4,dp)
    rec (15,46,6,6,dp)
    rec (17,48,2,2,co)
    rec (26,14,6,29,m1)
    rec (23,29,2,29,po)
    rec (27,47,6,6,m1)
    rec (29,26,6,6,m1)
    rec (29,26,6,6,po)
    rec (31,28,2,2,co)
    rec (0,0,36,26,cp)
FFIG
```

```
**********************************************
*                                            *
*      LUCIE file for an inverter            *
*                                            *
**********************************************

niv dn,dp,co,po,m1,cn,cp,vi,m2,pa,bo
FIG PINV
    rec (14,11,2,32,po)
    rec (0,26,16,2,po)
   *n device
    rec (0,0,29,23,cp)
    rec (0,3,29,6,m1)
    rec (4,3,6,16,m1)
    rec (4,3,6,6,dp)
    rec (6,5,2,2,co)
    rec (4,13,6,6,dn)
    rec (6,15,2,2,co)
    rec (10,14,10,4,dn)
    rec (20,12,6,30,m1)
    rec (20,12,6,6,dn)
    rec (22,14,2,2,co)
   *p device
    rec (0,33,29,21,cn)
    rec (4,36,6,6,dp)
    rec (4,36,6,15,m1)
    rec (6,38,2,2,co)
    rec (10,36,10,4,dp)
    rec (20,36,6,6,dp)
    rec (22,38,2,2,co)
    rec (0,45,29,6,m1)
    rec (4,45,6,6,dn)
    rec (6,47,2,2,co)
    rec (20,24,6,6,po)
    rec (22,26,2,2,co)
    rec (26,26,3,2,po)
FFIG
ef
```